VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky

DIPLOMOVÁ PRÁCE

2012 Bc. Jan Korpas

VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Dynamická simulace softwarových procesů

Software Process Dynamics Simulation

2012 Bc. Jan Korpas

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Jan Korpas**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Dynamická simulace softwarových procesů
Software Process Dynamics Simulation

Zásady pro vypracování:

Posláním práce je vytvořit simulační model softwarového procesů pomocí přístupů systémové dynamiky v simulačním nástroji a implementace tohoto přístupů v jazyce JAVA.
Cíle práce:
1. Prostudování a popis přístupu systémové dynamiky pro softwarové procesy.
2. Vytvoření simulačního modelu referenčního softwarového procesu.
3. Navržení rámce pro vytváření simulačních modelů v prostředí JAVA.
4. Implementace navrženého rámce.

Seznam doporučené odborné literatury:

ABDEL-HAMID, T. AND MADNICK, S. Software Project Dynamics: An Integrated Approach. Prentice Hall, 1991. ISBN 0138220409.
MADACHY, R.J. Software Process Dynamics. 2nd ed.: Wiley-IEEE Press, 2008. ISBN 0471274550.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Jan Kožusznik, Ph.D.**

Datum zadání: 18.11.2011
Datum odevzdání: 04.05.2012

VYSOKÁ ŠKOLA BÁŇSKÁ-TECHNICKÁ UNIVERZITA Fakulta elektrotechniky a informatiky * OSTRAVA *

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Čestné prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně a uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě dne: 3.5.2012

.....

Bc. Jan Korpas

Poděkování

Autor této práce by na tomto místě rád poděkoval svému vedoucímu Ing. Janu Kožuszníkovi, Ph.D., za jeho metodické vedení a cenné rady při konzultacích.

Také bych velice rád poděkoval svým rodičům a přítelkyni Kateřině za jejich srdečnou a trpělivou podporu poskytovanou nejen během psaní této práce, ale v průběhu celého studia na Fakultě elektrotechniky a informatiky.

Abstrakt

Tato práce je složena z teoretické a praktické části. Teoretická část pojednává o potřebě simulace v nesčetných oblastech každodenního života s důrazem na její využití v oblasti softwarového inženýrství, konkrétně při procesu změny softwarového systému. Z výčtu simulačních metod byla pro potřeby této práce zvolena systémová dynamika, jejíž principy a základní prvky jsou v této práci globálně prezentovány. Praktická část práce je nejprve zaměřena na vytvoření referenčního simulačního modelu vývoje změny software metodou systémové dynamiky. Závěr práce patří popisu implementace simulačního engine, jeho propojení s uživatelským rozhraním a následnému testování, jež ověří správnost, funkčnost sestrojeného engine a celého simulačního nástroje.

Klíčová slova

Systémová dynamika, simulace, simulační engine, model systémové dynamiky, Java, softwarové inženýrství, model změny software.

Abstract

This thesis deals with the simulation using the method of system dynamics especially in the software engineering area. It is divided into the theoretical and practical part. The theoretical one consists of the historical guide of the system dynamics and of definitions of the general terms connected with the topic. The first practical part contains the description of making the reference model of system dynamics simulation. The second practical parts describes the creation of the implementation of the simulation engine that was connected with GUI interface. It also consists of the result comparison between the created engine values and values that come from Vensim.

Key words

System dynamics, simulation, simulation engine, system dynamic model, Java, software engineering, change software model.

Seznam použitých symbolů a zkratek

API	- Application Programming Interface
GEF	- Graphical Editing Framework
GMF	- Graphical Modeling Framework
GMP	- Graphical Modeling Project
GNU	- Název projektu podporujícího vývoj svobodného software
GUI	- Graphical User Interface
HTML	- HyperText Markup Language
IBM	- International Business Machines Corporation
IDE	- Integrated Development Environment
JDK	- Java Development Kit
JDT	- Java Development Tools
JRE	- Java Runtime Environment
PHP	- Rekurzivní zkratka PHP: Hypertext Preprocessor
RCP	- Rich Client Platform
SLOC	- Source Lines of Code
UML	- Unified Modeling Language

Obsah

Seznam tabulek

Seznam obrázků

1 Úvod

Simulace je pojmem, se kterým se lze setkat v četných situacích každodenního života. Za synonymum k tomuto pojmu, které zároveň představuje jeho význam, můžeme pokládat napodobování rozličných dějů a různorodých situací. Proces vývoje software naopak můžeme pokládat za běžný pojem jen stěží, jelikož se s jeho realizací setkává pouze úzce profilovaná skupina softwarových vývojářů, a proto je pro širokou veřejnost jeho význam přinejmenším mlhavý.

V této práci autor směřuje k propojení obou těchto pojmů s cílem vytvořit ucelený náhled na problematiku možností tvorby simulačních modelů pro upgrade softwarových systémů. Tato problematika není nikterak nová, což bude možné později dokázat historickým exkurzem do této oblasti, a letitou praxí již byl mnohokrát prokázán přínos simulace právě pro oblast softwarového inženýrství. I přesto lze stále objevit pole působnosti, které nebyly zcela přesně popsány, nebo nebyly četné systémy doposavad vyvinuty. Nadto je nutné podotknout, že naprostá většina veškerých publikací zabývajících se problematikou simulací v systémovém inženýrství je cizojazyčná. Proto by i tato práce měla alespoň malým dílem přispět k osvětě tohoto tématu v českém prostředí. Tato práce nemá tendenci být pojata pouze jako kompilace vybraných zdrojů informací, ale má i obsáhlou praktickou část, ve které si autor klade za cíl vytvořit funkční referenční model vztahující se k předmětné problematice, následně uskutečnit implementaci simulačního engine, který bude propojen s uživatelským rozhraním.

Ve druhé kapitole této práce se seznámíme s důvody potřebnosti simulace v každém odvětví průmyslu či výroby a vývoje produktů a služeb, přednosti simulace vztáhneme na námi sledovanou problematiku procesu vývoje sofwarového systému. Z mnoha simulačních metod, které byly doposavad vyvinuty, byla pro tuto práci vybrána systémová dynamika, a to především k její vysoké variabilitě a mnohým přednostem, které si postupně představíme. Systémová dynamika a její využití v průběhu nedávné historie i současnosti budou taktéž představeny ve druhé kapitole.

Již název třetí kapitoly napovídá, že jejím obsahem bude náhled do základních principů a prvků systémové dynamiky. Znalost těchto principů a prvků je klíčovým předpokladem k pochopení této simulační metody a k následnému vytváření správně sestavených a funkčních modelů coby první etapy požadované simulace. Součástí této části práce bude také pojednání o zpětné vazbě a matematickém vyjádření systémové dynamiky.

Kapitola čtvrtá je tematicky úzce propojena s kapitolou předchozí, jelikož se věnuje samotnému vytváření zvoleného referenčního modelu vývoje změny softwarového systému. Nejprve jsou představeny základní vlastnosti simulačního modelu, které byly tvůrcem modelu zvoleny, poté je přistoupeno k samotnému modelování pomocí systémové dynamiky. Referenční simulační model je pro snazší pochopení rozdělen do tří částí, které mohou fungovat nezávisle na sobě. Nejprve se spojováním hladin, toků a dalších prvků modulu vytvoří základní řetězec, který představuje základní časovou posloupnost – etapizace procesu vývoje software do jednotlivých fází. Druhá část modelu se věnuje chybám, které se v rámci celého procesu vytvoří, posléze detekují a konečně dojde v různých fázích procesu k jejich odstranění. Poslední část modelu je

zaměřena na úsilí, které je v průběhu procesu či jeho jednotlivých fázích vynakládáno Aby však simulace co nejdetailněji odpovídala procesu vývoje změny softwarového systému, který se odehrává v realitě, je nezbytné, aby došlo k propojení všech tří částí modelu.

Poslední pátá část této práce je svým obsahem nejrozmanitější, jelikož pojímá celou druhou praktickou součást této práce – implementaci simulačního engine. Nejprve je pozornost zaměřena na technologie, kterých bylo v průběhu implementace využito. Mnohé z nich, kupříkladu programovací jazyk Java, jsou díky své notorické známosti a zároveň obsáhlosti popsány pouze velmi vágně s upozorněním na ty nejdůležitější údaje a principy. Kromě použitých technologií je popsána knihovna JEval, se kterou se v této práci taktéž pracovalo. Poté je v práci již přistoupeno k deskripci samotné architektury obsahující vyvinutý simulační engine, včetně jeho vztahu k uživatelskému rozhraní GUI, se kterým může být skrze interface propojen (ačkoliv na tomto rozhraní není závislý a může v budoucnu dojít k propojení s jiným rozhraním). Závěrečným pojednáním této páté kapitoly je testování, které ověří správnost vytvořeného simulačního engine. V rámci tohoto testování dojde nejen k otestování zvolených příkladů, ale také k otestování simulace vytvořeného referenčního simulačního modelu, který byl sestaven ve čtvrté kapitole této práce. Tímto dojde k myšlenkovému i faktickému propojení smyslu celé této práce.

2 Systémová dynamika a softwarový proces

2.1 Potřeba simulace v životě

Soudobá společnost zažívá i na počátku 21. století překotný vývoj civilizace, zejména na poli techniky a technologií. První známky rozvoje techniky a technologie lze spatřovat již v období průmyslové revoluce, dále lze za plodná období považovat zejména období obou světových válek, kdy vlivem militarizace došlo k obrovskému pokroku ve vědě a zbrojním průmyslu, jenž byl později využit i v oblastech civilních. Posléze se vývoj techniky v druhé polovině 20. století stále zrychloval a zdokonaloval, přičemž každá z oblastí vývoje zažívala období největšího rozkvětu v různých etapách tohoto období. Na poli vývoje počítačové techniky uveďme např. 50. léta a polovinu let šedesátých, kdy docházelo ke vzniku tzv. počítačů druhé generace a prvopočátkům používání operačních systémů, či následující období mezi lety 1965 – 1981, kdy docházelo ke vzniku tzv. počítačů třetí generace se zavedením interaktivních systémů (lze vzpomenout zejména na nejznámější řadu počítačů IBM 360).[1] Lze však již považovat za notorietu, že rychlost vývoje je právě na přelomu tisíciletí větší než kdykoliv předtím.

Konjunktura dnešní doby skýtá pozitivní i negativní dopady. Mezi klady nepochybně patří pokrok civilizace, možnost efektivnějšího využití zdrojů i postupů (od primární do terciární sféry ekonomiky) a v neposlední řadě snižování ceny výrobků s použitými technologiemi vlivem jejich rapidního zastarávání, což ocení zejména koneční uživatelé a spotřebitelé. S tímto však úzce souvisejí potíže, které provázejí oblast výroby a vývoje, jelikož produkty se stávají v krátkém časovém měřítku zastaralé a překonané konkurencí. Nepřetržitě se zvyšujících požadavků klientů a síla konkurence proto s sebou přináší neustálou potřebu vývoje produktů, postupů i strategií managementu ve všech oblastech výzkumu, výroby a obchodu. Při vývoji a inovaci svých produktů se společnosti neustále potýkají s klíčovou rolí časového faktoru, nemohou riskovat špatné rozhodnutí či volbu nesprávného postupu, jelikož důsledkem může být fatální ztráta konkurenceschopnosti a ztráta mnohdy složitě vydobyté pozice na trhu. Je proto přirozeným jevem, že společnosti hledaly cestu umožňující „nahlédnutí do budoucnosti“, tj. předvídání nejpravděpodobnějšího následku jejich konkrétních rozhodnutí. Prostředkem k dosažení požadavků se ukázalo být modelování situací a s tímto spojená počítačová simulace.

Proces simulace, jenž se skládá z návrhu simulačního modelu a následného provádění experimentu s konkrétními zadanými parametry, nachází v současnosti širokospektrální využití. Sehrává nezastupitelnou roli nejen na poli teorie a vědy (vědecké experimenty), ale také v každodenních i méně frekventovaných oblastech praktického života – předpovídání počasí,

[1] ZELENÝ, S., MANNOVÁ, B. *Historie výpočetní techniky*. 1. vyd. Praha: Scientia, 2006. 183 s. ISBN 80-86960-04-8. Kapitola 8, Třetí generace počítačů, s. 83-91.

trajektorie dopravních prostředků (zejména letadel), výzkum chemických molekulárních struktur, kosmologie, archeologie a mnoho dalších.[2] Jak již bylo zmíněno, své uplatnění našla simulace taktéž v oblasti vývoje a inovace produktů a technologií, jelikož výsledky simulací umožňují dospět k nejsofistikovanějšímu rozhodnutí, zvolení optimální strategie či rozvržení finančních, personálních i časových investic do konkrétních oblastí vývoje.

2.2 Simulace pomocí systémové dynamiky

Zájemcům o vytvoření simulačního modelu a následného vykreslení možného průběhu zvolené situace se otevírá velké množství simulačních metod, ze kterých je možno si vybírat ze všech úhlů pohledů tu nejpoužitelnější a nejvíce vyhovující. Nelze jednoznačně tvrdit, že nejnovější metody vždy zastíní metody dřívější, v mnoha případech hrají metody vyvinuté na přelomu 80. a 90. let důležitou roli v možnostech simulačního procesu a obohacují tak výčet metod modelování a simulace, z nichž lze selektovat tu pro daný případ nejoptimálnější. Lze však s jistotou tvrdit, že metody, jež lze uplatnit formou počítačové simulace, jsou zastoupeny ve výčtu nejčastěji využívaných metod s naprostou procentuální převahou, což pochopitelně koresponduje s technickou vybaveností jednotlivců a společností, které mají zájem o namodelování klíčových situací potřebných pro jejich business rozhodování. Při výběru metody je však nezbytné si uvědomit, že ne všechny metody mají univerzální použití, nýbrž jejich uplatnění je omezeno pouze na jistý okruh působnosti (kupř. technika, ekonomika, management apod.)

Pro orientační vstup do nesčetného výčtu si uveďme stručný demonstrativní výčet a popis vybraných metod modelování a simulace. Jednou z metod, která navzdory své letitosti (její vývoj spadá do 60. let 20. století) patří k nejpoužívanější zejména v technické praxi, je FEM (Final Elements Method) neboli metoda konečných prvků. Tato metoda založená na Lagrangeově principu byla uplatněna zejména v kosmonautice a leteckém průmyslu (pro zajímavost lze uvést její participaci na kosmickém výzkumu Apollo pro NASA). I v nynějším výrobním sektoru slouží ke zlepšování užitných vlastností výrobků či v oblastech strojírenství, hutnictví a elektrotechniky.[3] Stejně „archaickým“ (počátky jsou taktéž datovány do první poloviny 60. let 20. století), a přesto v mnoha odvětvích i na akademických půdách stále hojně užívanou simulační metodou jsou Petriho sítě založené na principech paralelismu a nedeterminismu.[4] Relativně úzce profilovanou metodou je BPMN, která díky svému přehlednému grafickému znázornění modelovaného problému napomáhá managementu společností k lepšímu pochopení svých vnitřních organizačních

[2] MADACHY, Raymond J. *Software Process Dynamics*. 1st printing, New Jersey: John Wiley & Sons, Inc., 2008. 601 s. ISBN 978-0-471-27455-1. Kapitola 1, Introduction and background, s. 4.
[3] KOLÁŘ, V. *FEM principy a praxe metody konečných prvků*. 1. vyd. Praha: Computer Press, 1997. 401 s. ISBN 80-7226-021-9. s 13-97.
[4] Petriho síť. *Wikipedie* [online]. Změněno 10.3.2012 [cit. 2012-03-20. Dostupné z: < http://cs.wikipedia.org/wiki/Petriho_s%C3%AD%C5%A5>.

struktur a interních pochodů a jejich změn, jež se mohou udát pod tíhou okolností a vnějších i vnitřních vlivů.[5]

Poslední ze zmíněných metod, systémové dynamice, bude věnována zvláštní pozornost, jelikož se jedná o metodu, která byla vybrána pro simulaci a modelování v následujících kapitolách této práce. Kořeny systémové dynamiky sahají, stejně jako u dvou z výše zmíněných metod, na přelom 50. a 60. let minulého století. Myšlenkovým otcem této metody se stal elektronický inženýr, profesor Jay W. Forrester. *„Jeho první článek zabývající se touto problematikou – „Industrial Dynamics – A Major Breakthrough for Decision Makers" – vyšel v Harvard Business Review v roce 1958 a zároveň se stal 2. kapitolou jeho pozdější knihy „Industrial Dynamics" z roku 1961, základního pilíře nové disciplíny."* [6] Ruku v ruce s myšlenkou rámcového pojetí systémové dynamiky byly na univerzitní půdě vyvíjeny skupinou akademiků kolem J. W. Forrestera modelovací jazyky – kompilátory diferenciálních rovnic. Nejprve byl sestaven jazyk SIMPLE (Simulation of Industrial Management Problems with Lots of Equations), který však byl zakrátko nahrazen jazykem DYNAMO (DYNAmic MOdels), který byl jeho vylepšenou verzí. Tento jazykový model se stal standardem po období následujících třiceti let.[7]

Ve svých začátcích byla systémová dynamika užívaná pro modelování otázek spjatých se řízením, managementem a ekonomickou situací. Vznikl zejména prototyp Modelu limitů růstu (Limits to Growth), který dokázal nasimulovat situace, ve kterých se vyjevily překvapivé faktory ovlivňující ekonomický růst obchodních společností.[8] Na sklonku 60. let však došlo k posunu využití i do jiných odvětví praktického života, a to zejména do oblasti urbanismu. Ve spolupráci s bývalým starostou Bostonu Johnem Collinsem sepsal Forrester knihu s názvem „Urban Dynamics". Model prezentovaný v této knize byl vnímán velmi rozporuplně a kontroverzně, jelikož nebylo všem politickým i nepolitickým uskupením vhod, aby se poukazovalo na špatné vedení a nedostatky města, jež mají za následek růst negativních faktorů (nezaměstnanost, kriminalita apod.).[9] *„Na základě Urban Dynamics se začaly rozšiřovat další modely, které se pokoušely o pochopení složitých dynamických sociálních systémů – „národní dynamika" (System Dynamics National Model), „světová dynamika" (World Dynamics), na kterých spolupracoval například i Římský klub."*[10] Období let 1970-1990 se neslo v duchu návratu k Modelu limitů růstu. Skupina autorů, jež byli v šedesátých letech studenty J. W. Forrestera publikovali postupně tituly „Limits to Growth" a „Beyond the Limits", které obsahovaly zejména myšlenky světové dynamiky

---

[5] *Object Management Group Business Process Model and Notation* . [online]. [cit. 2012-04-15]. Dostupné z: <http://www.bpmn.org/>

[6] MILDEOVÁ, S., VOJTKO, V. a kol. Systémová dynamika. 2. vyd. Praha: Oeconomica, 2008. 150 s. ISBN 9708-80-245-1448-2.s. 9.

[7] *System dynamics.* [online]. [cit. 2012-04-16]. Dostupné z: <http://www.systemdynamics.org/DL-IntroSysDyn/start.htm>

[8] MILDEOVÁ, S., VOJTKO, V. a kol. Systémová dynamika. 2. vyd. Praha: Oeconomica, 2008. 150 s. ISBN 9708-80-245-1448-2.s. 9.

[9] *System dynamics.* [online]. [cit. 2012-04-16]. Dostupné z: <http://www.systemdynamics.org/DL-IntroSysDyn/start.htm>

[10] MILDEOVÁ, S., VOJTKO, V. a kol. Systémová dynamika. 2. vyd. Praha: Oeconomica, 2008. 150 s. ISBN 9708-80-245-1448-2.s. 9.

a opětně rozšířily povědomí o této modelovací metodě v jiných průmyslových a socioekonomických odvětvích.

Ani nedávná minulost a současnost na simulaci pomocí systémové dynamiky nezapomínají, ačkoliv byly od dob jejího vzniku vyvinuty mnohé velmi zdařilé modelovací metody. Dopomáhá k simulacím ekonomických otázek zejména ve Spojených státech amerických a Kanadě. Mimo jiné je také významným prvkem na akademických půdách, kde pomáhá osvětlovat nejen příčiny novodobých a aktuálních problémů, ale zpětně pomáhá simulovat historické milníky (např. světovou krizi v třicátých letech 20. století), a tak napomáhá pochopení provázanosti ekonomiky, výroby, sociální sféry a dalších faktorů ovlivňujících společnost. I přes zdánlivou celistvost metody probíhají v systémové dynamice změny a upgrady, které pomáhají reflektovat vývoj nejen ostatních metod, ale i technologií a nových společenských jevů.[11]

2.3 Simulace v oblasti softwarového inženýrství a vývoje softwarových systémů

Mezi odvětví, která mohou těžit ze simulace možného postupu v otázkách, které se dostaly do pomyslného rozcestníku nadcházejícího vývoje, patří i softwarové inženýrství a oblast výzkumu a vývoje softwarových systémů. Potřeba náhledu do budoucnosti, který simulace skrze své modely nabízí, je v překotném vývoji nových systému v posledních letech stále markantnější, jelikož je to právě počítačová technika a software, který ovlivňuje chod ve všech sférách ekonomiky od těžby po terciární sektor služeb.

V mnoha ohledech lze v procesu vývoje softwarových systému nalézt paralely s jinými oblastmi vývoje výrobků či služeb, které simulaci a modelování při svém působení využívají jako jednu z možných taktik v konkurenčním boji. I softwarový systém lze totiž vnímat jako konečný produkt, jenž je tvořen skrze invenci konkrétního pracovního personálu v obchodních společnostech. Je proto předvídatelné, že i v oblasti softwarového inženýrství vyvstávají nejpalčivější otázky zejména ve sféře vedení a plánování managementu, a to jak na úrovni vrcholového managementu celé obchodní společnosti, tak v jednotlivých vývojových odděleních. Jejich tematickou náplní bývá v naprosté většině případů otázka časové a finanční náročnosti a s tím spjatá problematika vstupního kapitálu, kvantity a kvality personálního obsazení a dalších interních i externích faktorů, které do vývoje produktů přímo či zprostředkovaně zasahují a konečný produkt ovlivňují. Kromě zmíněných témat, která jsou na první pohled pochopitelná, lze pomocí simulace osvětlit i otázky vývoje softwarových systémů, jež jsou relativně obtížněji uchopitelné – např. chybovost, kvalifikace či celkové nebo částečné úsilí.

I přes podobnost a úzkou provázanost s jinými oblastmi vývoje a výroby jiných produktů či služeb má softwarový proces přece jen svá specifika, která mohou do velké míry ovlivnit průběh

[11] *System dynamics.* [online]. [cit. 2012-04-18]. Dostupné z: <http://www.systemdynamics.org/DL-IntroSysDyn/start.htm>

vývoje produktu a jeho finální podobu. Jedním ze specifik je časová stránka vývoje nových či upravených softwarových systémů. Vlivem obrovské dynamiky trhu je poptávka po nových softwarových systémech stále vzrůstající a vývoj tak musí probíhat ve velmi rychlém tempu s cílem reagovat na požadavky zákazníků v pomyslném časovém limitu a pochopitelně rychleji než konkurenční společnosti. Dalším specifikem vývoje softwaru je jeho častá tvorba na míru zákazníkovi dle konkrétní zakázky. Společnosti zabývající se softwarovým inženýrstvím tedy nemohou spoléhat pouze na kreativitu a invenci svých pracovníků, ale jsou do velké míry vázány požadavky svých klientů či potenciálních klientů. Jejich požadavky musejí splnit co nejpřesněji a také v době co nejkratší (časové hledisko ostatně bývá jedním z klíčových požadavků zákazníka). S faktem, že softwarové systémy jsou velice různorodé a málokdy se v nově vyvinutém produktu opakuje nezanedbatelné procento prvků z vývoje jiných systému, úzce souvisí i poslední specifikum, a to omezená možnost vzít si zpětnou vazbu z tvorby identického či velmi podobného produktu, který byl vyvíjen dříve. Společnostem tudíž zůstává jen omezená možnost využívání best practices. Z výše uvedeného vyplývá, že právě simulace je pro vývoj software neocenitelným přínosem, který pomáhá eliminovat nesnáze, jež vznikají právě v souvislosti s uvedenými specifiky softwarového inženýrství.

Potřeba vyrovnání se s časovým hlediskem vývoje software byla jedním ze stěžejních témat již od samotného vzniku softwarového inženýrství. Jedním z průkopníků softwarového inženýrství byl Fred Brooks, jenž je v povědomí širší veřejnosti spojován zejména s vývojem řady počítačů IBM 360.[12] Právě Brooks se stal v roce 1975 autorem jednoho z klíčových výroků souvisejících s problematikou času při vývoji nového produktu v rámci softwarového inženýrství, který je znám pod pojmem Brooksův zákon: „*Přidání řešitelské kapacity u zpožděného softwarového projektu způsobí jeho další zpoždění.*“ [13] Poukázal tak na úzkou provázanost časového faktoru a změny kapacity lidských zdrojů, konkrétně pak na fakt, že myšlenka navýšení personálu při vývoji nového softwarového systému se vždy nemusí ukázat jako efektivní. Pokud je totiž nová posila doplněna do týmu po určitém časovém mezníku, může doba potřebná pro její zaškolení způsobit ještě výraznější zpoždění finální podoby produktu. Tuto teorii ještě podrobněji rozpracovalo duo softwarových inženýrů Tarek Abdel-Hamid a Stuart Madnik na počátku 90. let minulého století. Ti na základě výpočtů a modelace zavedli pravidlo, že k zaškolení nového zaměstnance až do jeho úplné asimilace do projektového týmu je zapotřebí jedna čtvrtina pracovní kapacity zkušeného (tj. již zcela zapracovaného) člena týmu, přičemž průměrná doba asimilace je 20dní. Jinak řečeno, nový pracovník, který přistupuje jakožto úplný nováček do projektu, je po dobu 20 dní zaučován jednou čtvrtinou zkušeného člověka, než je schopen plnohodnotně nastoupit do projektu.[14]

---

[12] *Fred Brooks*. Wikipedie [online]. Změněno 23.4.2012 [cit. 2012-04-24]. Dostupné z: < http://en.wikipedia.org/wiki/Fred_Brooks>.
[13] *Softwarové inženýrství*. Wikipedie [online]. Změněno 12.2.2012 [cit. 2012-04-22]. Dostupné z: < .http://cs.wikipedia.org/wiki/Softwarov%C3%A9_in%C5%BEen%C3%BDrstv%C3%AD >
[14] MADACHY, Raymond J. *Software Process Dynamics*. 1st printing, New Jersey: John Wiley & Sons, Inc., 2008. 601 s. ISBN 978-0-471-27455-1. s. 19.

Právě Brooksův zákon se stal jedním ze stěžejních stavebních kamenů pro modelování a simulaci v oblasti vývoje softwarových systémů, jelikož prokázal, jak důležitou roli hraje předvídání dopadu rozhodnutí na finální podobu projektu a okolností s ním spjatých. Je pochopitelné, že vztah časového faktoru a lidských zdrojů není jediným problematickým místem vývoje v softwarovém inženýrství, tudíž může simulace nastoupit v nesčetných případech různých modelovaných situací.

2.4 Simulace vývoje softwarového systému pomocí systémové dynamiky

Simulace a modelování může i v oblasti vývoje softwarového inženýrství probíhat skrze různé simulační metody, a to včetně těch metod, které jsou nastíněny v kapitole 2.2. Tak jako u jiných oblastí, ve kterých se přistupuje k predikci nejasných záležitostí pomocí simulování, je nezbytné provést pečlivou selekci, na jejímž konci bude volba nejoptimálnější simulační metody, která bude co nejpřesněji korespondovat s konkrétními požadavky, jež jsou na výsledky simulace kladeny.

Systémová dynamika, jakožto jedna z možných simulačních metod, byla v oblasti vývoje softwarových systémů, byla poměrně dlouho opomíjená. Ačkoliv její kořeny sahají, jak bylo taktéž podrobněji popsáno v kapitole 2.2, do druhé poloviny 50. let minulého století, o jejím využití pro softwarové inženýrství se začalo uvažovat až o 30 let později. Za průkopníka tohoto využití by mohl být označen profesor Tarek Abdel-Hamid, který již od roku 1986 působí na Naval Postgraduate School v Kalifornii jako profesor informačních věd a dynamiky systému.[15] Právě jeho disertační práce, jejíž dokončení směřovalo do roku 1984, obsahovala simulační model, který byl vytvořen na principech systémové dynamiky J.W. Forrestera. Abdel-Hamid tento model dále postupně rozpracovával společně s profesorem Stuartem Madnickem, jenž je již od roku 1972 spjatý s akademickou půdou na Massachusetts Institute of Technology.[16] Výsledkem této plodné kooperace je publikace s názvem Dynamika softwarového projektu (Software Project Dynamics) vydaná v roce 1991, která se právem stala milníkem v oblasti simulace vývoje softwarových systémů pomocí systémové dynamiky.

Profesoři Abdel-Hamid a Madnick nebyli samozřejmě jedinými akademiky, kteří se tématem simulace v softwarovém inženýrství pomocí systémové dynamiky zabývali. Systémová dynamika se na přelomu 80. a 90. let 20. století dostala do popředí zájmu známého amerického počítačového vědce a nositele mnoha ocenění Gerryho Weinberga, který své myšlenky s tímto tématem zakomponoval do knihy Quality Software Management: Anticipating Change. Vol. 1: Systems Thinking (1992). Je zajímavou souhrou náhod, že ačkoliv tato publikace vznikla téměř ve stejné době jako Dynamika softwarového projektu, autoři navzájem o svých záměrech nevěděli.

[15] *Naval Postgraduate School, Monterey, California* [online]. [cit. 2012-03-29] Dostupné z: <http://faculty.nps.edu/vitae/cgi-bin/vita.cgi?p=display_vita&id=1023567788>.
[16] *Stuart Madnick* [online]. [cit. 2012-03-29]. Dostupné z: <http://web.mit.edu/smadnick/www/home.html>

Weinberg se ve své knize primárně zaměřuje na parafrázi Brooksova zákona, věnuje se důsledkům špatných rozhodnutí a špatně zvolených strategií. Stal se taktéž zastáncem zpětné vazby v procesu, když tvrdí, že manažeři projektu musí především vše pečlivě naplánovat s využitím všech předchozích zkušeností, předvídat důsledky a poté jednat tak, aby se co nejvíce přiblížili tomu, co si v začátcích naplánovali.[17] Ve zkratce lze uvést demonstrativní výběr dalších teoretiků, akademiků i odborníků z praxe, kteří se v rámci své práce a výzkumu zabývali systémovou dynamikou a jejím uplatněním při simulaci v oblasti softwarového inženýrství - patřili či patří mezi ně kupř. Marc. I Kellner, Bill Curtis nebo Chuen-Lung Chen.

Na závěr uvedeme několik informací o profesoru Raymondu J. Madachym, jenž v současnosti stejně jako profesor Abdel-Hamid působí na Naval Postgraduate School v Kalifornii. Ačkoliv je zmíněn jako poslední, jeho přínos pro simulaci softwarového procesu metodou systémové dynamiky je zásadní. Představíme-li si pomyslnou linii klíčových osob pro téma, jež bude dále prezentováno v této práci, na počátku stojí J.W. Forrester coby zakladatel systémové dynamiky, dalšími jsou T. Abdel-Hamid a S. Madnick, jejichž přínos spočívá v propojení systémové dynamiky se softwarovým inženýrstvím, ale až poslední, R. J. Madachy, vdechl do této problematiky život formou praktického využití a přiblížil tvorbu simulačního modelu odborné i širší veřejnosti. Při svém působení ve vědecké sféře Madachy čerpal nejen z dlouholetého působení v praxi na postech technického vedoucího projektů ve společnostech zabývajících se vývojem softwarových systémů (např. společnosti C-Bridge Institute či Xpert Cost Group), ale i z jeho profesní dráhy kantora, který mnoho let působí na několika prestižních univerzitách. Byl si proto vědom toho, že model dvojice Abdel-Hamid a Madnick byl sice doveden téměř k dokonalosti, ale praxe potřebuje jeho variabilitu a schopnost se v aktuálním čase přizpůsobit vyvstalým požadavkům. To doposavad modely neumožňovaly, jelikož byly vždy překládány jako pouze hotový produkt s návodem k jeho užívání. Madachy však trval na tom, že cílem všeho by mělo být především naučit běžné uživatele vytvořit si vlastní model, který bude optimalizován na konkrétní požadavky. Ve svém stěžejním díle *Software Process Dynamics (2008)* představil kompletní metodiku tvorby simulačních modelů pomocí systémové dynamiky s podrobným představením jednotlivých funkcí, jejich významu a důležitosti pro model jako celek.[18]

Madachyho tvorba se stala velkou inspirací i návodem pro tvorbu této práce. Na jejím podkladě byl vytvořen nejen simulační model referenčního softwarového procesu, jenž bude představen v kapitole 4, ale jeho myšlenky byly stěžejní pro implementaci softwarové dynamiky v jazyce Java, jež je náplní poslední kapitoly této práce.

---

[17] MADACHY, Raymond J. *Software Process Dynamics*. 1st printing, New Jersey: John Wiley & Sons, Inc., 2008. 601 s. ISBN 978-0-471-27455-1. s. 10.
[18] MADACHY, Raymond J. *Software Process Dynamics*. 1st printing, New Jersey: John Wiley & Sons, Inc., 2008. 601 s. ISBN 978-0-471-27455-1. s. 9.

3 Prvky a principy systémové dynamiky a jejich uplatnění v softwarovém procesu

3.1 Základní principy systémové dynamiky

Každá simulační metoda je vystavěna na základních principech, které se promítají do celého rámce a zásadně platí pro všechny oblasti dané metody. Nejinak je tomu i u systémové dynamiky, která svým založením spadá do skupiny těch metod, které jsou primárně založeny na modelování – tj. vytváření modelu a následné provedení simulace tohoto modelu s konkrétně zadanými parametry. V tomto modelování se klade důraz především na zpětné vazby, konkrétně na tvorbu sítě vzájemně propojených uzavřených zpětnovazebních smyček (o zpětnovazebních smyčkách v kapitole 3.3). *„Východiskem systémově dynamického přístupu je, že mnohé systémy se chovají v rozporu s očekáváním. Mnoho lidí si neumí představit efekt nelinearity a efekt zpětné vazby zásahu ve složitých systémech, většinou dramaticky podceňují setrvačnost systému, která vede k nesprávné závěrečné politice. Zobecňujeme lineárně, ale chování je většinou více než složité vlivem závislostí mezi proměnnými a smyčkou zpětné vazby, kde výstup systémové komponenty má podstatný vliv jako vstup v budoucnosti. Dodatečná složitost je představována existencí akumulací a zpoždění.“* [19]

Podstatným principem systémové dynamiky je jeho komplexnost spočívající v možnosti kombinovat jak hledisko kvalitativní tak kvantitativní, není potřeba se zaměřovat pouze na jeden aspekt – systémová dynamika se nezaměřuje na konkrétní predikce, ale na ucelené vzorce chování. Proměnné jsou podrobeny neustálé změně, v závislosti na hodnotách dalších proměnných. Zjednodušeně řečeno, vše je systémem zpětných vazeb propojeno se vším, což umožňuje dokonalou simulaci reálných situací.[20]

Mezi další základní principy, které jsou důležité hlavně v pohledu softwarové dynamiky, patří obrovská variabilita a možnost propojení s ostatními metodami a nástroji. Z praktické stránky lze vyzdvihnout tuto variabilitu spočívající v možnosti modelování v řadě výkonných softwarových systémů (pro příklad uveďme Vensim, ProSim, AnyLogic, iThink aj.). Z tohoto důvodu je systémová dynamika metodou velmi přístupnou pro široké spektrum jednotlivců i společností, jelikož každý má zkušenosti s jiným systémem.

---

[19] MILDEOVÁ, S., VOJTKO, V. a kol. *Systémová dynamika*. 2. vyd. Praha: Oeconomica, 2008. 150 s. ISBN 9708-80-245-1448-2. s. 10.
[20] MADACHY, Raymond J. *Software Process Dynamics*. 1st printing, New Jersey: John Wiley & Sons, Inc., 2008. 601 s. ISBN 978-0-471-27455-1. s. 147.

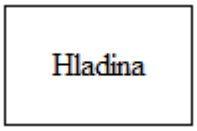
3.2 Prvky modelování v systémové dynamice

Modelace vždy probíhá skrze typické prvky, kterými je ta která metoda specifická. Díky těmto prvkům zasvěcení také vždy na první pohled na model rozpoznají, prostřednictvím které metody bude konkrétní model simulován. Každý prvek je definován nejen svým názvem, ale také specifickým grafickým znázorněním, kterým bývá ve většině případů v modelu znázorňován. Pochopení základních prvků a jejich vlastností je klíčovým momentem celé simulace, proto je nutné vždy věnovat důkladnou pozornost jejich studiu. Dokonalá znalost prvků a pohotové přiřazení jednotlivých faktorů do správné skupiny prvků je jedinou možnou cestou pro správné uchopení modelace pomocí systémové dynamiky.

3.2.1 Hladiny

Hladinami jsou označovány tzv. stavové proměnné, tj. proměnné hodnoty modelu, které charakterizují stav systému - představují akumulaci změn systému za časový okamžik. Jejich hodnoty jsou měněny v každém časovém kroku a jsou určovány rozdílem přítoků a odtoků směřujících do respektive z hladiny. *„V rovnovážném stavu by se celkové přítoky do hladiny rovnaly celkovým odtokům z hladiny. Hladina však umožňuje, aby se přítoky a odtoky lišily.“*[21] Hladina se vyznačuje těmito nejmarkantnějšími vlastnosti:

- můžeme je chápat jako jakousi paměť uchovávající informace o systému včetně minulých stavů systému
- oddělují úrovně toků
- jsou zdrojem zpoždění v systému
- grafické znázornění hladin ve většině modelovacích nástrojů je reprezentováno obdélníkem, jak je vidět na obrázku Obr. 1.



Obr. 1: Ukázka označování hladin

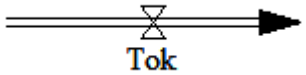
V oblasti softwarového inženýrství lze poukázat na celou řadu typických představitelů hladin. Uveďme krátký výčet exemplárních zástupců, jenž bývají nejčastěji v modelech zastoupeny – množství aktuálně pracujících zaměstnanců, náklady na softwarový proces, celkové vynaložené úsilí, počet chyb v software, podíl na trhu atd. Z tohoto výčtu by mohlo dojít k nabytí mylného dojmu, že hladiny vždy představují hodnoty kvantitativní vyjadřující určité množství. Jak bylo již řečeno, systémová dynamika pracuje s kvantitativními i kvalitativními aspekty, a proto i

[21] MILDEOVÁ, S., VOJTKO, V. a kol. *Systémová dynamika*. 2. vyd. Praha: Oeconomica, 2008. 150 s. ISBN 9708-80-245-1448-2. s. 62.

v hladinách můžeme najít abstraktněji vyjádřené proměnné vyjadřující kvalitativní aspekt, jako jsou kupříkladu kvalita, spolehlivost, zabezpečení, funkčnost či uživatelská použitelnost.

3.2.2 Toky

S hladinami jsou neodmyslitelně spjaté tzv. toky, jejichž úrovně jsou hladinami odděleny. Chceme-li si přiblížit a demonstrovat vzájemný vztah toků a hladin na paralele z každodenního života, můžeme si vše představit na jednoduchém modelu vany, kde samotná vana a voda v ní reprezentuje hladinu, zatímco přítok a odtok vody do vany představuje jednotlivé toky a jejich úrovně. Z tohoto příkladu můžeme snadno vyvodit, jak toky pracují. *Jestli hladiny jsou „stavy" systému, toky jsou „akce".*[22] Toky činností ovlivňují hodnoty hladin v každém časovém kroku tak, že je navyšují nebo snižují v závislosti na výsledku rozdílu vstupních a výstupních toků. Toky jsou jedinými prvky v modelu, které mohou měnit hodnotu hladin. Grafická podoba toků ve většině modelovacích nástrojů je pak reprezentována šipkou s ventilem, jak je patrné z obrázku Obr. 2.



Obr. 2: Ukázka označování toků

Nejčastějšími představiteli toků v softwarovém inženýrství jsou kupříkladu alokování a dealokování zaměstnanců, programování, generování chyb, tvorba návrhu softwarového systému apod.

Rozeznat od sebe toky a hladiny by mělo být základní dovedností každého tvůrce modelu. Zvlášť v začátcích vytváření modelu může právě tato selekce představovat pro většinu tvůrců nemalé problémy. Ačkoliv se mohou objevit výjimky, pár níže uvedených pravidel může dopomoci ke správné identifikaci jednotlivých faktorů a jejich přiřazení do správné skupiny[23]:

- Hladiny obvykle představují podstatná jména vyjadřující stavy (kvalita, úsilí, počet chyb aj.), zatímco toky lze rozpoznat pomocí sloves či takových podstatných jmen, která vyjadřují úkony nebo činnosti (tvorba, programování, nárůst aj.)
- Pokud by se čas hypoteticky zastavil (např. pokud bychom pořídili aktuální snímek systému), uvidíme pouze hladiny, toky zmizí a nezobrazí se nám
- Hladiny vysílají informace o stavu systému do zbytku systému

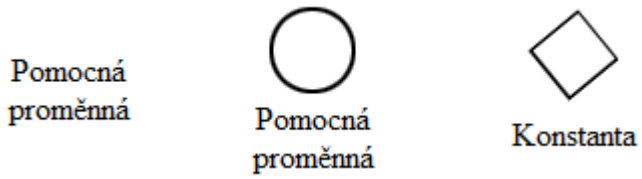
---

[22] MILDEOVÁ, S., VOJTKO, V. a kol. *Systémová dynamika*. 2. vyd. Praha: Oeconomica, 2008. 150 s. ISBN 9708-80-245-1448-2. s. 62.
[23] *System dynamics* [online]. [cit. 2012-04-21]. Dostupné z: <http://www.systemdynamics.org/DL-IntroSysDyn/start.htm>.

3.2.3 Pomocné proměnné a konstanty

Dalšími hojně zastoupenými prvky v modelech jsou pomocné proměnné. Tyto proměnné mají dvojí základní funkci, a to pomáhat rozvést strukturu jednotlivých hladin a toků do podrobnějších detailů a dále slouží jakožto převodník vstupních hodnot na výstupní.[24] Pomocné proměnné mohou být definovány algebraickými výrazy složenými z dalších pomocných proměnných, hladin, toků nebo konstant. Neméně častou podobou je jejich definování prostřednictvím funkcí, které jsou charakteristické pro nějaký proces, jež je touto pomocnou proměnnou reprezentován. Naproti tomu konstanty jsou takovými prvky, jejichž hodnota je v průběhu simulace modelu neměnná. V modelech většinou najdou konstanty využití pro definici počátečních hodnot ostatních prvků.

V modelech se lze často setkat se situací, kdy jsou pomocné proměnné či konstanty označené pouze názvem, který není doprovázen žádným grafickým znázorněním. Pokud se už ke grafické podobě přistoupí, bývají pomocné proměnné a konstanty označeny shodným způsobem pomocí kružnice nebo pouze jejich názvem. Jak je patrné z obrázku Obr. 3, v případě konstant se můžeme setkat s dalším možným znázorněním, a to symbolem kosočtverce.



Obr. 3: Ukázka označování pomocných proměnných a konstant

Pomocné proměnné i konstanty mohou být v oblasti softwarového inženýrství zastoupeny stejnými představiteli, což vyplývá z možnosti dosadit do jejich hodnot konkrétní číslo (konstantu) či rovnici (proměnnou). Z dlouhého výčtu možných představitelů můžeme vyjmenovat kupříkladu míru výskytu chyb, plánované množství splněných úkolů, procento dokončení práce, produktivita složek apod.

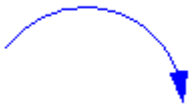
3.2.4 Spoje

Je logické, že simulace vytvořeného modelu může úspěšně proběhnout jen za předpokladu, že jsou veškeré hladiny, toky a pomocné proměnné či konstanty správně vzájemně propojeny. K tomuto propojení systémová dynamika využívá posledních z prvků – spojů. Právě prostřednictvím spojů jsou přenášeny vstupní hodnoty pro výpočetní a rozhodovací úkony. Spoje taktéž slouží k přenosu zpětnovazebních hodnot z proměnných (obvykle z hladin nebo z

[24] MADACHY, Raymond J. *Software Process Dynamics*. 1st printing, New Jersey: John Wiley & Sons, Inc., 2008. 601 s. ISBN 978-0-471-27455-1. s. 57.

pomocných proměnných) směrem k tokům a funkčním proměnným, které tyto hodnoty použijí pro své výpočty.

Grafická podoba spojů v modelech je v naprosté většině případů znázorněna jednoduchou šipkou. V některých odnožích systémové dynamiky se však můžeme setkat s třemi typy znázornění, jež korespondují s rozdělením spojů do tří typů: informační – zajišťuje přenos informací mezi proměnnými, inicializační – nastavuje počáteční hodnoty hladin a zpožďovací – zajišťuje přenos takových informací, u nichž nastala prodleva.[25] Konkrétní obecné znázornění i grafická podoba jednotlivých typů je nejlépe patrná z Obr. 4



Obr. 4: Ukázka označování spojů

3.2.5 Zdroje a výpusti

Veškeré funkční prvky systému již byly obsáhnuty v předchozím výkladu této práce, přesto se při tvorbě modelu setkáme ještě s dalšími prvky, kterými jsou zdroje a výpusti (které se často označují anglickým ekvivalentem *sinks*). Tyto prvky se od všech výše zmíněných podstatně liší v absenci jakékoliv funkčnosti v modelu, při simulaci s nimi není nikterak pracováno. Přesto mají v modelu své nezastupitelné místo, jelikož určují hranice modelovaného systému jako celku, jsou jakýmisi hraničními body celého modelu. Zdroje poskytují počáteční parametry, které vstupují do námi modelovaného systému. Jelikož zdroje stojí vně našeho systému, není již předmětem našeho zájmu, jak se v nich tyto počáteční parametry vytvářejí, vstupují tak jako faktická notorieta. Příkladem zdrojů mohou být rozličné požadavky přicházející zvenčí, jiná oddělení v obchodní společnosti, která nezasahují do samotného vývoje software, outsourcingové společnosti apod. Naopak výpusti jsou výstupními subjekty simulace modelu, tedy opakem zdrojů. Stejně jako u zdrojů se nadále nezajímáme o další nakládání s těmito výsledky, protože již do našeho modelu nepatří. Reprezentanty výpustí mohou být například výsledný software, klienti, kterým je tento software dodáván, či zaměstnanci, kteří opouštějí modelovaný proces.

Zdroje a výpusti mají v modelech systémové dynamiky taktéž přiřazený specifický symbol, jenž je společný pro oba tyto prvky. Jeho podoba bývá označována jako stylizovaný symbol mraku – viz Obr. 5.

[25] MILDEOVÁ, S., VOJTKO, V. a kol. *Systémová dynamika*. 2. vyd. Praha: Oeconomica, 2008. 150 s. ISBN 9708-80-245-1448-2. s. 64.

Obr. 5: Ukázka označování zdrojů a výpustí

3.3 Zpětná vazba

Pro pochopení globálního fungování simulace pomocí systémové dynamiky je nezbytné pojmout široké povědomí o zpětných vazbách, jelikož model vytvořený na základě systémové dynamiky je založen právě na vzájemných souvislostech mezi uzavřenými zpětnovazebními řetězci. Zpětná vazba tak definuje jednu ze základních relací mezi jednotlivými prvky modelovaného sytému. Zpětnovazební procesy jsou jevem, který je možno často pozorovat v běžných systémech z reálného světa, kdy výstupní procesy takovéhoto systému zpětně ovlivňují jeho vstupní procesy. Velmi zdařilý příklad takového systému najdeme v oblasti softwarového inženýrství v procesu testování. Do testování vstupuje software s chybami, jež jsou v testech nalezeny a následně programátory opraveny. Těmito opravami však mohou vzniknout opět nové chyby, které budou následně odhaleny a znovu opraveny. Oprava nalezených chyb v software tak zpětně ovlivňuje vznik dalších chyb.

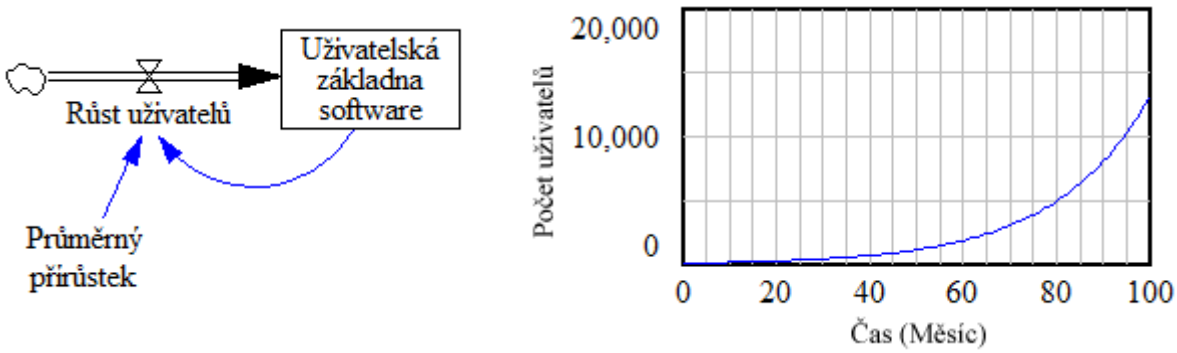
Podle vlivu na modelovaný systém, dělíme zpětné vazby na dva typy – pozitivní a negativní. Na závěr si představíme také zpoždění systému, který může být považován za zvláštní druh zpětné vazby v modelu.

3.3.1 Pozitivní zpětná vazba

Pozitivní zpětná vazba vytváří v modelovaném systému kladné (tj. rostoucí) hodnoty – kupříkladu můžeme zmínit pozitivní růst, zvětšování odchylek či zesilování působení změn. Říkáme, že je sebeposilující, a proto je někdy označována jako posilující zpětná vazba. Růst, jenž se díky pozitivní zpětné vazbě generuje, je exponenciální. Příkladem může být úročení finančních prostředků na bankovním účtu, kdy čím větší obnos na účtu je, tím větší se připisují úroky. Je nutné si však uvědomit, že kladné rostoucí hodnoty mohou být při simulaci také ty hodnoty, které v běžném životě vnímáme jako záporné - klesající. *„Příkladem toho, jak stupňování zpětné vazby může urychlit pokles, je růst prodejů akcií v důsledku poklesu důvěry a současný pokles cen akcií, což vede k dalšímu poklesu důvěry.“*[26] Na tomto příkladu je tedy patrné, že podstata pozitivní zpětné vazby je ono urychlení, i když se jedná o urychlení poklesu.

[26] MILDEOVÁ, S., VOJTKO, V. a kol. *Systémová dynamika*. 2. vyd. Praha: Oeconomica, 2008. 150 s. ISBN 9708-80-245-1448-2. s. 56.

Konstrukci pozitivní zpětné vazby v systémové dynamice a příslušný graf jsou znázorněny na obrázku Obr. 6. Tento obrázek představuje další příklad zpětnovazební smyčky, tentokrát z oblasti námi sledovaného softwarového procesu.



Obr. 6: Konstrukce a graf pozitivní zpětné vazby

3.3.2 Negativní zpětná vazba

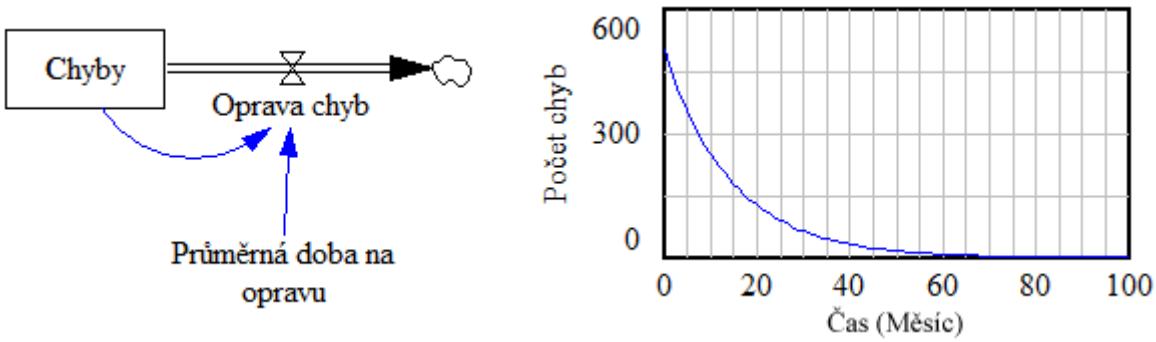
V negativní zpětné vazbě na rozdíl od pozitivní zpětné vazby rozdíly nenarůstají a, naopak se v modelovaném systému vytváří rovnováha. Lze tak říci, že negativní zpětná vazba se snaží přivést celý systém do výsledného cílového stavu. V případě, že existuje rozdíl mezi skutečným a požadovaným stavem, je jejím prostřednictvím provedena korekce směřující k dosažení cíle. Jednoduchým příkladem je proces jedení, které redukuje míru hladu. Nárůst hladu způsobí, že daný člověk více jí. Naopak, čím více toho dotyčný sní, tím se jeho hlad sníží.[27]Míra korekce vlivem negativní zpětné vazby je odvislá od velikosti rozdílu mezi skutečným a požadovaným stavem systému, kdy markantní rozdíl vyvolá velké reakce a menší rozdíl má tendenci vyvolávat reakce menšího rázu.

Po dosažení rovnováhy systému se negativní zpětná vazba vždy snaží kýžený cílový stav udržovat v tomto ideálním stavu i do budoucna, čehož dosahuje pomocí kladení odporu veškerým změnám, které by mohly znamenat odchýlení se z požadovaného stavu. Negativní zpětná vazba tak generuje opak exponenciálního růstu, jelikož sledovaná hodnota nejprve výrazně kleslá a poté se limitně blíží nule.

Při vývoji softwarového systému se ve všech fázích setkáváme s výskytem chyb a nutností jejich opravy. V tomto jevu můžeme nalézt typický příklad chování, které odpovídá negativní zpětné vazbě, a to závislost průměrné doby potřebné na opravu chyb vzhledem k jejich počtu Konstrukci této vazby a její výsledný graf zobrazuje obrázek Obr. 7.

---

[27] *System dynamics* [online]. [cit. 2012-04-23]. Dostupné z: <http://www.systemdynamics.org/DL-IntroSysDyn/start.htm>.



Obr. 7: Konstrukce a graf negativní zpětné vazby

3.3.3 Zpoždění

Zpoždění není typickým příkladem zpětné vazby, jedná se však o jednu z nejdůležitějších součástí zpětnovazebních systémů, která má se zpětnou vazbou mnoho vlastností společných, a proto bývá k tématu zpětných vazeb v literatuře často přiřazována a někdy bývá dokonce označována jako třetí zpětnovazební typ po boku pozitivní a negativní zpětné vazby. Chce-li tvůrce modelu vytvářet simulační systém, který bude věrně odrážející realitu, neměl by na vliv zpoždění za žádných okolností zapomínat.

Časové zpoždění se nachází v každém modelovaném procesu z reálného světa. Běžně se s ním setkáme také během vývoje software, například v procesech schvalování změn, přijímání nových zaměstnanců, řešení vzniknuvších problémů a tak dále. Velkou měrou se na zpoždění podílí komunikace členů vývojového týmu. Je již známým faktem, že je přímá úměra mezi počtem lidí, kteří mají spolu komunikovat, a délkou (tj. zpožděním) doby, ve kterém se dostaví výsledek dané komunikace. Časové zpoždění můžeme vyjádřit jako průměrnou dobu, po kterou daná jednotka setrvá v hladině, než z ní bude odtokem přesunuta dále. Odtok je pak definován podílem hladiny a časového zpoždění. Tato formulace má také tu výhodu, že odtok nemůže způsobit, aby dospěla hladina do záporných hodnot.

3.4 Matematické vyjádření systémové dynamiky

Závěrečným principem, který v této práci bude představen, je zapisování simulačních modelů pomocí matematických rovnic. Obecně lze říci, že modely vytvořené každou modelační metodou bývají zapsány pomocí matematických rovnic, které vyjadřující strukturu daného modelovaného jevu. Simulační modely vytvoření pomocí systémové dynamiky jsou pak zapisovány systémem dynamických rovnic, které popisují chování modelu v čase. Aby byl model

zapsán korektně, mely by rovnice splňovat základní vlastnosti, tj. existenci řešení, jednoznačnost řešení a spojitou závislost na vstupních údajích modelu.[28]

K vytváření dynamiky v modelech systémové dynamiky jsou coby dynamické rovnice používány rovnice integrální a diferenciální. Matematickou podobu modelu simulovaného systémovou dynamikou tak můžeme vyjádřit následující diferenciální rovnicí:

$$x'(t) = f(x, p),$$

kde *x* je vektor hladin, *p* je množina parametrů a *f* je tzv. přechodová funkce. Takto vyjádřená rovnice představuje pouze základní dynamiku modelu. *Abychom tuto dynamiku mohli projektovat, je třeba ještě specifikovat, co přesně znamená „změna x(t)“.*[29] Ta nám umožňuje zachytit změnu hladin v libovolném čase *t*, kterou můžeme zapsat pomocí integrální rovnice takto:

$$Hladina = Hladina_0 + \int_0^t (Přítok - Odtok)\, dt.$$

Tato rovnice vyjadřuje stav hladiny *Hladina* za čas *0* až *t*. Její hodnota je určena z původní hladiny $Hladina_0$, která je navýšena o rozdíl *Přítoku* a *Odtoku* v čase *dt*. Hodnota *dt* představuje zvolený inkrementační čas každého kroku.

Tento popis slouží pouze pro ilustraci matematického vyjádření systémové dynamiky a pro práci s modely systémové dynamiky je plně dostačující. Výhodou dnešních modelovacích nástrojů je, že jsou modely vizualizovaný a není potřeba, aby tvůrce modelu přišel do styku s diferenciálními či integrálními rovnicemi.[30] Z tohoto důvodu je bližší pojednání o těchto rovnicích za hranicemi této práce.

[28] POSPÍŠIL, Zdeněk. Dynamické systémy a systémová dynamika. In Dynamika systémů a udržitelnost. 1. vyd. Brno: Nadace Partnerství, 2007. od s. 25-68, 44 s. Soubor učebních textů. s. 38.
[29] Tamtéž
[30] MADACHY, Raymond J. *Software Process Dynamics*. 1st printing, New Jersey: John Wiley & Sons, Inc., 2008. 601 s. ISBN 978-0-471-27455-1. s. 56.

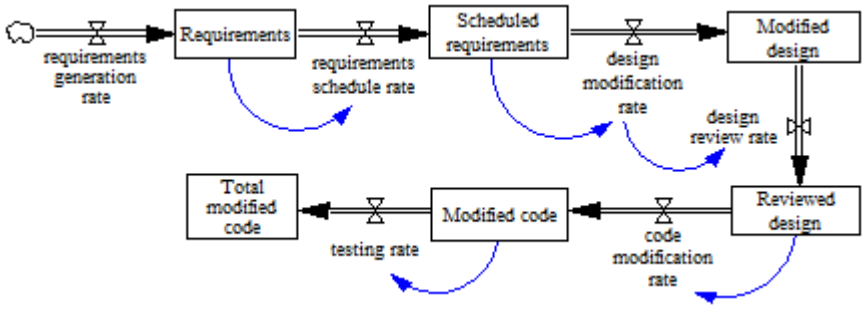
4 Simulační model referenčního softwarového procesu

4.1 Problematika vývoje a změny software

Zrod nového softwarového systému může vycházet ze dvou přístupů. Prvním z nich je vývoj zcela nového systému, který není nikterak ovlivněn předchozími produkty. Ačkoliv se však obchodní společnosti mnohdy chlubí systémem zcela novým, je nutné podotknout, že je ve většině případů poněkud obtížné nevyužít zpětné vazby a ponaučení se z předchozího vývoje, a proto systémy i přes svou deklarovanou novost vždy obsahují ve větší či menší míře prvek rukopisu svého předchůdce. Druhým přístupem je v předchozí větě nastíněný upgrade stávajícího softwarového systému, což ve většině případů představuje vývojovou změnu a vylepšování systému v průběhu jeho životního cyklu. Právě tento přístup umožňuje reflektování nedostatků a chyb, a proto může snáze dosáhnout plného uspokojení konečného uživatele systému, jenž prošel předmětnou softwarovou změnou, která zásadně přináší kýžené vylepšení.

Námi prováděná simulace se bude dotýkat druhého přístupu ke zrodu nového softwarového systému, tedy změny a přepracování stávajícího software. Změnu softwarového systému je třeba vždy chápat jako velmi rozsáhlý a komplikovaný proces, jehož modelování je složitě uchopitelné, jelikož se lze velmi rychle ztratit při modelaci souvislostí mezi jednotlivými podprocesy systému, ze kterých se tento proces skládá. Výčet takových podprocesů je v průměrném systému vývoje vskutku široký a s největší pravděpodobností by zabral několik stránek.

Pro základní pochopení simulace změny software je nutné seznámit se se základním řetězcem, jenž představuje stěžejní body vývoje softwarového systému. Jak lze vyčíst z obrázku Obr. 8, těmito stěžejními body jsou generování požadavků, plánování požadavků, modifikace návrhu, revize návrhu, modifikace kódu a konečné testování.



Obr. 8: Základní řetězec procesu změny software

Z výše uvedeného je jasné, že takto provedená simulace by byla použitelná jen ve zcela izolovaném a inertním prostředí, kde by simulovaný proces nebyl nikterak ovlivňován dalšími

okolnostmi. Chceme-li však simulovat vývoj změny softwarového systému co nejvěrohodněji, musíme nastavit další faktory, které co nejblíže odpovídají realitě a vývojové praxi.

Jen stěží si lze představit, jak velké množství faktorů na výsledek simulace působí, a to právě prostřednictvím výše zmíněných podprocesů. Namátkou lze uvést zlomkový výčet nejdůležitějších podprocesů, jako je specifikace požadavků, plánování a přidělování požadavků, návrh, revize návrhu, schválení návrhu, kódování, testování a oprava chyb, nasazení, najímání zdrojů apod. Každý z podprocesů by mohl být reprezentován samostatným dosti velkým modelem, který participuje na modelu hlavním, čímž dochází k rozšiřování základního modelu z obrázku Obr. 8.

Z výše uvedeného vyplývá, že komplexní model obsahující všechny podprocesy se pro účely vytvoření referenčního modelu, jenž má za úkol přiblížit a osvětlit simulační proces v systémové dynamice, příliš nehodí. Orientace v takto rozvinutém modelu by byla složitá, a tak by byl celý proces těžce pochopitelný. Pro naše potřeby bude uvažován model změny software do jisté míry zmenšený, který nám i přesto poskytne dostatečný náhled na modelovanou problematiku a vyjádří vše podstatné.

4.2 Popis referenčního modelu

Námi vybraný referenční model, jenž bude níže popisován a rozebírán, představuje, jak již bylo zmíněno, pouze část celkového a značně rozsáhlého modelu simulace změny softwarového systému, konkrétně se jedná rozvinutý základní řetězec doplněný o simulaci generování chyb a vynaloženého úsilí. Pokud by měl být model rozšířen do podoby, která se co nejvíce přibližuje reálnému vývoji softwarového systému, bylo by nutné do něj doplnit řadu aspektů a okolností, jež mohou do procesu změny software vstoupit, a tak konečný výsledek simulace podstatně ovlivnit. Jedná se například o procesy fluktuace zaměstnanců, získávání zkušeností zaměstnanců postupem času, působení únavy na produktivitu zaměstnanců aj.

Dříve, než přistoupíme k samotnému vytváření zvoleného referenčního modelu, je potřeba definovat několik základních vlastností modelovaného problému. Toto definování je podmínkou sine qua non, jelikož je nezbytné vytvořit si rámec, na jehož základě budeme schopni vytvořit model, který bude odpovídat našim představám a bude představovat simulaci požadovaného problému. Potřebnými základními vlastnostmi jsou: účel modelu, ohraničení modelu, klíčové proměnné a v neposlední řadě popis požadovaného chování modelu. Pro účely definování existuje celá řada metod a postupů, proto záleží pouze na tvůrci modelu, kterou cestu si zvolí.

4.2.1 Účel modelu

Účelem zvoleného referenčního modelu je umožnění simulace situací, které mohou nastat v průběhu procesu, jenž začíná plánováním úkolů a končí testováním změněného software. Na základě námi zadaných parametrů všech veličin budeme schopni sledovat čas potřebný pro dokončení tohoto procesu, počet chyb, které se ve fázích modifikace návrhu a kódování vyskytnou, a jaké úsilí budeme v jednotlivých časových úsecích aktuálně potřebovat.

Simulace nám tak poskytne velmi cenná data, jež poslouží k dosažení co největší efektivity při všech procesech, jež jsou součástí změny software. Není třeba zdůrazňovat, že tato data by mohly být významnými výsledky i v globálním měřítku celistvé simulace vývoje změny software, tedy při integraci dalších faktorů a aspektů jakožto souboru podprocesů do celistvého modelu kompletního vývoje změny softwarového systému. S největší pravděpodobností by však došlo k modifikaci těchto hodnot právě vlivem nově přidaných uzavřených podprocesů a zpětnovazebních smyček. Nad rámec této práce by bylo jistě velice zajímavé sledovat, jakým způsobem a v jaké míře dochází k ovlivňování námi nasimulovaných výsledků vlivem dosazení dalších, v referenčním modelu vynechaných, faktorů.

4.2.2 Chování modelu

Další ze základních vlastností je chování modelu. Tvůrce modelu si při definici této vlastnosti musí představit, jak daný proces probíhá v reálném prostředí zaběhnuté praxe a jaké faktory na tento proces působí. Tuto představu musí následně co nejvěrněji aplikovat do vytvářeného modelu, tak, aby byly logicky zachovány všechny postupy a etapy vývoje softwarového systému a model tak byl obrazem reality v co největších detailech.

Prezentovaný referenční model části procesu změny softwarového systému začíná, tak jako podstatná většina procesů nejen v oblasti softwarového inženýrství, plánováním úkolů. V této první fázi projektoví manageři obdrží seznam požadavků, jejichž splněním dojde ke změně původního softwarového systému. Obdržením požadavků ze strany klienta tak vzniká prostor pro splnění prvního, ale zároveň zcela zásadního úkolu projektového managementu – naplánování celého procesu změny softwarového systému do podoby, jež je klientem požadovaná. Čas, jenž je k této fázi zapotřebí, je přímo úměrně závislý na výkonnosti projektových manažerů.

Po naplánování celého procesu změny software se přistoupí k následující fázi, kterou je modifikace návrhu, jež je zajišťována týmem zvolených návrhových inženýrů. Modifikace návrhu je první kreativní fází celého vývoje nového systému, jelikož je nutné vytvořit metodiku jednotlivých kroků vedoucích k požadované změně software. Vstupní informací, která je podkladem pro tuto invencí členů týmu návrhových inženýrů, je seznam úkolů předaný projektovými manažery, kteří jej vytvořili v první fázi projektu. Výstupem činnosti vyvíjené při modifikaci návrhu je pochopitelně přepracovaný návrh, tj. návrh pozměněný oproti původnímu návrhu, který fungoval před požadovanou změnou. Také v tomto případě je nezbytné do modelu zadat hodnotu výkonnosti dotčených pracovníků, která definuje objem úkolů, který jsou tito schopni zvládnout za zvolenou časovou jednotku. Aby bylo možné se změněným návrhem dále pracovat, je potřeba provést jeho revizi a odhalit v něm případné nedostatky, čímž se dostáváme do další fáze vývoje, která je v referenčním modelu uvažována. V tomto případě nepočítáme se situací, kdy jsou nedostatky nalezené během revize posílány znovu do modifikace návrhu k přepracování. Pro zjednodušení celého procesu a jeho modelu se o revizi, stejně jako o modifikaci návrhu, stará tým návrhových inženýrů, přičemž jejich vyšší vytíženost je zohledněna v celkovém počtu členů týmu a efektivitě práce.

Finální podoba revidovaného modifikovaného návrhu změny softwarového systému je následně předána do fáze kódování neboli modifikace kódu, kde je na základě změnových

požadavků vycházejících z modifikovaného návrhu kód původního software modifikován. Ačkoliv se úkonům v rámci fáze kreativita mnohdy upírá, je nutné přiznat, že vytvoření správného a funkčního zdrojového kódu není jen ryze technickou dovedností, ale naopak vyžaduje jistou dávku tvořivosti. Nadto je třeba mít na paměti, že právě správně sepsané kódování je klíčovým prvkem fungování celého softwarového systému. Rychlost, s jakou je fáze kódování vykonávána, určuje zaprvé počet nasazených programátorů coby členů vývojového týmu a dále množství úkolů, které jsou tito programátoři schopni za zvolenou časovou jednotku zpracovat.

Výstupní modifikovaný kód je v závěru celého procesu změny software poslán k testování. Podobně jako v případě revize návrhu, také zde nebudeme uvažovat zpětné odstranění nalezených chyb ve fázi modifikace kódu. Ty budou zaznamenány a jejich oprava bude realizována v jiné části modelu, jejíž popis následuje dále v textu. Testování je poslední fází, kterou jsme zahrnuli do našeho referenčního modelu.

Jelikož se nepohybujeme v izolovaném prostředí, ale naopak veškeré procesy jsou postihovány chybami lidského selhání či strojů, je také v tomto případě nutné brát v úvahu vznik chyb, které se v průběhu vývoje softwarového systému mohou vyskytnout. Ty se budou logicky objevovat pouze ve fázích modifikace návrhu a modifikace kódu, přičemž jejich množství bude určeno hustotou chyb připadajících na jeden úkol. Chyby lze rozdělit na dva typy. Prvním typem je taková skupina chyb, které týmy návrhových inženýrů a programátorů odhalí během procesů modifikace návrhu respektive modifikace kódu, tudíž je možné je přímo v těchto fázích odstranit. Druhým typem jsou chyby, které se při vší snaze všech členů vývojového týmu nepodaří během modifikace návrhu či modifikace kódu nikým detekovat. Nedetekované chyby ve fázi modifikace návrhu proto budou ovlivňovat chybovost v následující fázi modifikace kódu, jelikož je zřejmé, že chybný návrh zvyšuje pravděpodobnost vzniku chyb v ostatních fázích, které tento návrh používají. Chyby, jež nebudou detekovány ve fázi modifikace kódu, budou nalezeny a odstraněny až v závěrečných testech celého softwarového systému.

Poslední sledovanou metrikou v našem referenčním modelu je výsledné úsilí všech dotčených pracovníků, které bylo vynaložené na modelovaný proces. Jeho výsledná hodnota zahrnuje dílčí úsilí, které bylo postupně vynaložené ve všech fázích základního řetězce, krom tohoto zahrnuje i úsilí sloužící k nalezení a odstranění chyb při modifikaci návrhu, modifikaci kódu a během závěrečného testování. Tato metrika nám taktéž umožňuje sledování aktuálně vynakládaného úsilí na proces ve zvoleném čase.

4.2.3 Klíčové proměnné modelu

Další nezbytnou součástí modelu, jež je zapotřebí zvolit si před započetím s tvorbou simulačního modelu a následnou simulací, jsou proměnné, které budou v daném modelu figurovat. V našem referenčním modelu se nachází několik základních proměnných, které se největší měrou podílejí na vytváření dynamického chování v modelu, nebo mají pro danou simulovanou situaci důležitou vypovídající hodnotu. Takové proměnné můžeme označit pracovně označit jako klíčové, čímž je odlišíme od zbývajících proměnných, jejichž dopad na model není tak výrazný, a popsat si blíže jejich specifika.

Námi modelovaný proces změny softwarového systému začíná přijetím změnových požadavků od klientů, které jsou vytvářeny za hranicemi modelu, a proto je pro nás důležitou hodnotou pouze jejich celkový počet. Změnové požadavky do modelu vstupují po dílčích částech skrze příslušný generující tok, který pracuje s hodnotou reprezentující celkový počet požadavků, jež je potřeba zpracovat a následně naplánovat.

Ačkoliv nejsou toky přímo prvky modelu, jež jsou zahrnutý mezi proměnné, neměli bychom je zapomenout označit jakožto velmi importentní prvky, a to zejména toky generující chyby ve fázích modifikace návrhu a modifikace kódu. Tyto zmíněné toky vygenerují z celkového počtu všech zpracovaných úkolů v obou z uvedených fázích určitý počet chyb, který je určen na základě chybovosti každého jednotlivého úkolu. V této souvislosti je pak nepochybně nutno považovat za klíčovou hodnotu proměnné, která je vypočítávána na základě celkového počtu nedetekovaných chyb ve fázi modifikace návrhu. Tato proměnná pak ovlivňuje tok generování chyb během modifikace kódu tak, že zvyšuje poměr chybovosti v této fázi.

Jak již bylo zmíněno v předchozí kapitole, v modelu změny software budeme také sledovat vynakládané úsilí v jednotlivých fázích celého procesu, které bude akumulováno do jediné výsledné hladiny. O plnění této hladiny se stará tok generující hodnoty, jež jsou rovny součtu vynakládaného úsilí aktuálně prováděných fází procesu změny software.

4.3 Vytvoření referenčního modelu

Jelikož bylo v kapitole 4.2 pojednáno o všech základních pojmech, se kterými systémová dynamika pracuje při simulaci procesu změny softwarového systému, byly zmíněny všechny podstatné vlastnosti, které je potřeba si zvolit před započetím s tvorbou simulačního modelu, a definovali jsme hlavní klíčové proměnné modelu, můžeme přistoupit k samotnému vytvoření grafické reprezentace námi zvoleného referenčního modelu pomocí systémové dynamiky v simulačním nástroji. Právě grafická podoba modelu umožňuje usnadnění chápání jednotlivých prvků a vazeb, jež byly dříve v této práci zmíněny v teoretické rovině.

Aby byl postup tvorby modelu snadněji pochopitelný, rozdělíme jej na tři části. První část bude reprezentovat základní řetězec procesů změny software, druhá část bude představovat chyby, které vzniknou v první části a nakonec třetí část, jež bude zaznamenávat veškeré vynaložené úsilí z první a druhé části. Všechny tyto části by mohly obstát jako samostatné modely bez závislosti na ostatních částech.

4.3.1 Simulační nástroj Vensim

Chceme-li využít výhod, jež nám skýtá simulace a modelování, v plném rozsahu je nutné protřídit si širokou škálu simulačních nástrojů (jako demonstrativní výčet lze uvést kupříkladu iThink, AnyLogic, Powersim Studio nebo Vensim) a zvolit si pro konkrétní model ten nevhodnější. Při výběru můžeme zohlednit nejen fakt, zda je užití vybraného nástroje poskytováno zdarma, je nabízena free trial licence, nebo se jedná o nástroj zcela zpoplatněný, ale také zejména je nezbytně důležité prozkoumat, zda možnosti simulace v daném nástroji odpovídají naším vstupním požadavkům.

Pro simulaci zvoleného referenčního modelu byl vybrán simulační nástroj Vensim, jenž je dobře známým a hojně využívaným nástrojem pro realizaci modelů systémové dynamiky, tudíž nám formou výsledného modelu poskytne relevantní odpovědi na stanovené oblasti našeho zájmu. Jeho velkou předností je, že je mimo jiné nabízen ve verzi Vensim PLE, která je určena pro vzdělávací a nekomerční účely, a proto je poskytována bezúplatně. Tato verze sice nedisponuje všemi vlastnostmi, jako jiné vyšší verze tohoto nástroje, ale pro naše edukační účely je bohatě dostačující.

4.3.2 Základní řetězec

Základní řetězec nám byl již blíže představen na obrázku Obr. 8. Připomeňme, že představuje pouze nejzákladnější posloupnost procesu vývoje softwarových systémů bez přídatných ovlivňujících faktorů. Vstupem do tohoto řetězce je celkový počet požadavků (*Requirements*), které následně projdou fází plánování týmem projektových manažerů, ve které se přemění na úkoly (*Tasks*), které jsou dále předány dále k dalšímu zpracování. Ačkoliv je základní řetězec zcela nerozvětvený, představuje celý proces změny softwarového systému, ve kterém hladiny akumulují požadavky či úkoly, které je potřeba v jednotlivých fázích zpracovat, a toky mezi hladinami tyto úkoly distribuují do dalších fází procesu. Výstupem je hladina akumulující všechny naplánované a poté zpracované úkoly. Aby nám poskytl ucelené informace o základním průběhu toku, obohatili jsme tento řetězec o několik důležitých prvků modelu, které budou v této části práce blíže představeny. Grafickou podobu takto rozšířeného základního řetězce lze nalézt na konci této podkapitoly.

Vstupní parametry

Vstupní parametry představují hodnoty, které definují chování celého základního řetězce. Tyto parametry jsou hodnotami proměnnými, které je při provádění simulace možné volit dle potřeby a na základě toho celý proces změny softwarového systému optimalizovat.

Vstupní parametry lze rozdělit na dva typy, z nichž pak vyplývají i měrné jednotky, ve kterých jsou jednotlivé parametry udávány. Prvním typem jsou pracovníci, kteří určují, kolik lidských zdrojů je v dané fázi potřeba (měrná jednotka *Person*). Druhým typem je produktivita, která definuje množství úkolů, které jsou pracovníci denně schopní zpracovat (měrná jednotka *Task/Person*). Samostatným parametrem, který nespadá ani do jednoho typu, je objem požadavků *Job size* představující celkový rozsah změny software. Tento parametr je udáván v měrných jednotkách přijatých požadavků (*Requirement*). Všechny vstupní parametry přehledně znázorňuje tabulka Tab. 1.

Vstupní parametr	**Měrná jednotka**	**Vstupní parametr**	**Měrná jednotka**
Job size	Requirement	Code modification productivity	Task/Person
Tasks scheduling productivity	Task/Person	Programmers	Person
Project managers	Person	Testing productivity	Task/Person
Design modification productivity	Task/Person	Testers	Person
Design engineers	Person		

Tab. 1: Základní řetězec – vstupní parametry

Hladiny

Kromě hladin reprezentujících počty požadavků či úkolů čekajících na zpracování, jsou v základním řetězci také dvě hladiny, které pro fáze modifikace návrhu a modifikace kódu akumulují celkový počet úkolů, které byly v těchto fázích zpracovány během celého procesu změny software. Z výčtu akumulovaných úkolů lze poté vycházet při stanovení dalších vlastností modelu, které závisejí na celkovém množství zpracovaných úkolů. Konkrétní užití vždy záleží na požadavcích tvůrce modelu a jím požadovaných finálních hodnot modelu a simulace jako celku. V našem referenčním modelu lze najít konkrétní využití hladiny kumulativního modifikovaného návrhu pro metriku chyb v procesu, což bude blíže rozebráno v kapitole 4.3.3 popisující druhou část modelu rozšířenou o problematiku chyb. Za zmínku dále stojí také hladina fungující na stejném principu jako první dvě zmiňované, která akumuluje celkový počet přijatých požadavků, a v tomto modelu nám umožňuje kontrolovat, zda byly přijaty již všechny požadavky ze stanoveného počtu.

Tyto tři zmíněné hladiny jsou označeny jako kumulativní. Seznam hladin včetně jejich měrných jednotek, jež jsou zahrnuty v tomto základním řetězci simulačního modelu, uvádí následující tabulka Tab. 2.

Hladina	**Měrná jednotka**	**Hladina**	**Měrná jednotka**
Requirements	Requirement	Reviewed design	Task
Cumulative requirements	Requirement	Modified code	Task
Scheduled tasks	Task	Cumulative modified code	Task
Modified design	Task	Total modified software	Task
Cumulative modified design	Task		

Tab. 2: Základní řetězec – hladiny

Toky

Toky, které figurují v základním řetězci referenčního modelu zajišťují předávání požadavků a úkolů mezi jednotlivými fázemi procesu změny software. Rychlost s jakou jsou požadavky a úkoly předávány, je závislá na vstupních parametrech každého toku. Nejdůležitějším tokem této části je *requirements generation rate*, který představuje generování požadavků na změnu. Stanovený objem požadavků je po částech předán v několika krocích simulace. Počet kroků je definován hodnotou přímo v rovnici toku. Kompletní seznam všech toků v této části modelu popisuje následující tabulka Tab. 3.

Tok	**Měrná jednotka**	**Tok**	**Měrná jednotka**
requirements generation rate	Requirement/Day	design review rate	Task/Day
cum requirements generation rate	Requirement/Day	code modification rate	Task/Day
tasks schedule rate	Task/Day	cum code modification rate	Task/Day
design modification rate	Task/Day	testing rate	Task/Day
cum design modification rate	Tasks/Day		

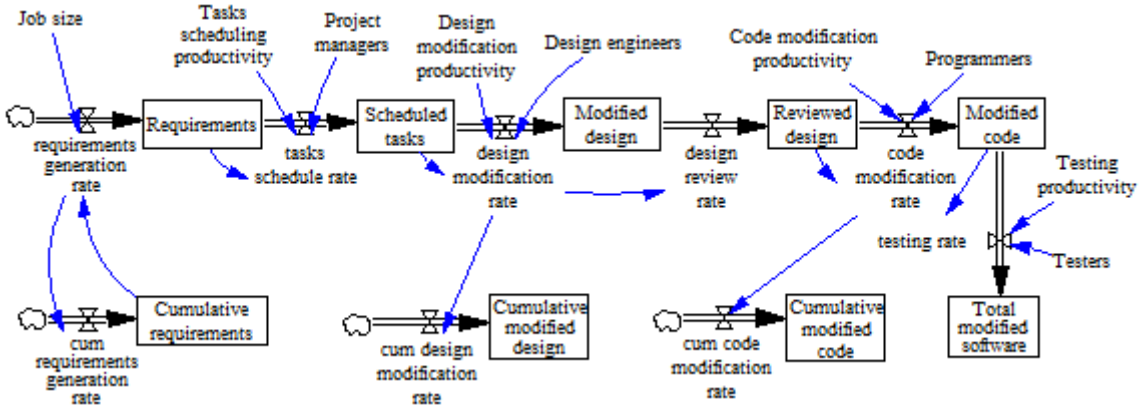
Tab. 3: Základní řetězec – toky

Výstup

Za výstupní prvky části základního řetězce referenčního modelu bychom mohli považovat každý tok či hladinu, protože pokud bychom si po skončení simulace zobrazili grafy jejich průběžných hodnot, byly by vždy nějak charakteristické a poskytovaly nám ty klíčové informace celého průběhu simulace. Měli-li bychom upozornit na ty nejzajímavější, mohli bychom zmínit hladinu *Total modified software*, jež představuje průběh celkově dokončené změny software.

Pro druhou část modelu obsahující výskyt chyb v průběhu procesu vývoje, jejíž popis bude následovat, se pak toky *design modification rate* a *code modification rate* stanou vstupními a z jejich hodnot bude vycházeno pro další výpočty.

Grafická reprezentace výše popsaného základního řetězce modelu je zobrazena na obrázku Obr. 9.



Obr. 9: Základní řetězec referenčního modelu

4.3.3 Generování chyb

V procesu změny softwarového systému je vždy nutno počítat se selháním lidského faktoru ve vývojovém týmu a s tím spojeným vznikem chyb. Z tohoto důvodu dochází v této části práce k rozšíření do značné míry idealizovaného modelu změny softwarového systému právě o zaznamenávání chyb, které vzniknou ve fázích modifikace návrhu a modifikace kódu. Jak bylo výše zmíněno, pro stanovení počtu chyb se využívá toků *design modification rate* a *code modification rate*. Z každého úkolu, který těmito toky projde, vygenerujeme příslušný počet chyb, jenž je určen poměrem chybovosti připadající na jeden úkol. V tomto procesu se snažíme zachytit také situaci, kdy dochází k ovlivňování chybovosti během kódování skrze chyby návrhu. Tímto tudíž dochází k propojení obou bloků, tj. chyby návrhu a chyby kódování, které by při neuvědomění si těchto souvislostí mohly být chápany jako samostatné.

Vstupní parametry

V této části modelu jsou vstupními parametry úkoly a poměry chybovosti, pomocí nichž je určován výsledný počet chyb, se kterým bude dále pracováno. Úkoly jsou poskytovány dvěma výše popsanými toky a poměry chybovosti představují konstanty *Design modification errors density* a

Code modification errors density. Pro stanovení počtu chyb ve fázi modifikace návrhu jsou použity již zmíněný tok a konstanta, kdežto ve fázi kódování je počet chyb navíc určen pomocí proměnné *Design errors density in code*, kterou ale nelze považovat za vstupní parametr, jelikož její hodnota je stanovena až v průběhu simulace. Seznam všech vstupních parametrů v této části modelu zobrazuje tabulka Tab. 4.

Vstupní parametr	**Měrná jednotka**
Design modification errors density	Error/Task
Design error detecting efficiency	Detected Error/Error
Code modification errors density	Error/Task
Code error detecting efficiency	Detected Error/Error
Error density	Error/Task

Tab. 4: Generování chyb – vstupní parametry

Hladiny

Základními hladinami této části modelu jsou *Design errors* a *Code errors*, které akumulují chyby vzniklé ve fázích modifikace návrhu a modifikace kódu. V případě první zmíněné hladiny jsou chyby generovány jen z jednoho zdroje – zmíněné fáze modifikace návrhu, naproti tomu výsledný počet chyb druhé hladiny se částečně skládá také z chyb, které nebyly detekovány během fáze modifikace návrhu a byly tak přičteny k chybám vzniklým ve fázi modifikace kódu. Chyby z těchto dvou hladin jsou dále děleny na ty, které byly pracovníky nalezeny během provádění modifikace návrhu a kódu, jež jsou reprezentovány hladinami *Detected design errors* a *Detected code errors* a na chyby, které nebyly nikým zaznamenány a ty jsou pak akumulovány v hladinách *Escaped design errors* a *Escaped code errors*. Hladina *Escaped code errors* navíc představuje chyby, které jsou na konci celého procesu změny software odhaleny v testech. Konečná hodnota hladiny *Errors corrected in test* odpovídá celkovému počtu nalezených chyb ve fázi testování. Seznam všech hladin v této části modelu zobrazuje tabulka Tab. 5.

Hladina	**Měrná jednotka**	**Hladina**	**Měrná jednotka**
Design errors	Error	Detected code errors	Error
Detected design errors	Error	Reworked code errors	Error
Reworked design errors	Error	Escaped code errors	Error
Escaped design errors	Error	Errors corrected in test	Error
Code errors	Error		

Tab. 5: Generování chyb - hladiny

Toky

V první části, která se týká generování chyb z procesu modifikace návrhu, je stěžejním tokem *design modification errors generation rate*, jenž určuje počet chyb vygenerovaných během této fáze v každém kroku simulace. Druhá část, jež představuje chyby vzniklé v procesu modifikace kódu, obsahuje dva důležité toky, které jsou společně přičítány k hladině *Code errors*. Jedná se o toky *code modification errors generation rate* a *design error amplification rate*. Druhý

zmíněný tok navyšuje hladinu *Code errors* o chyby, které nebyly nalezeny a odstraněny v první části. Posledním důležitým tokem je *testing correction rate* představující odstranění chyb, které nebyly do té doby odhaleny, a proto s nimi bylo pracováno až ve fázi testování. Seznam všech toků v této části modelu zobrazuje tabulka Tab. 6.

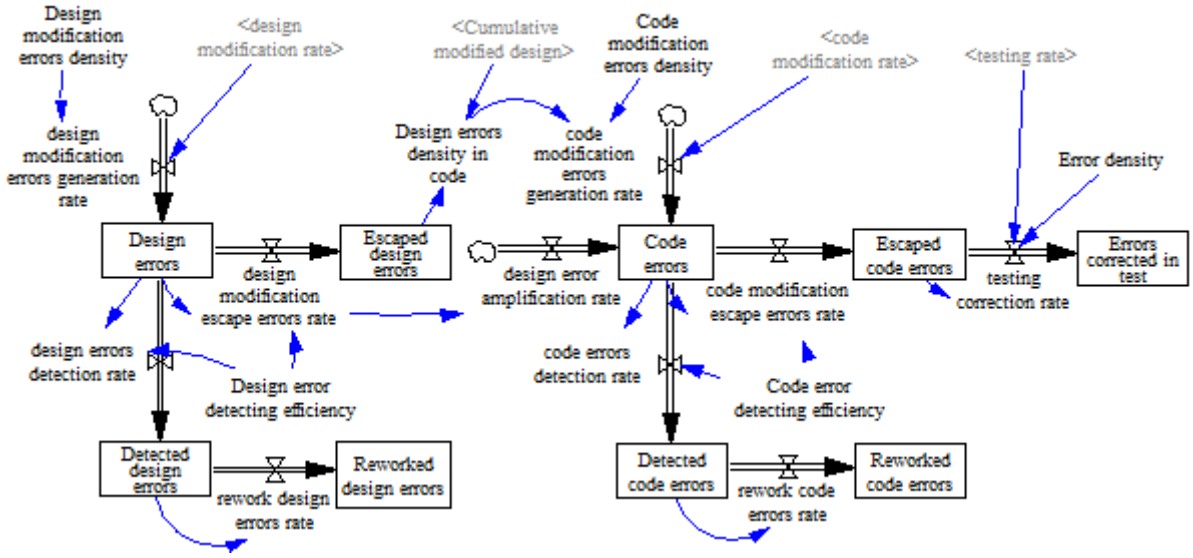
Tok	**Měrná jednotka**	**Tok**	**Měrná jednotka**
design modification errors generation rate	Error/Day	design error amplification rate	Error/Day
design errors detection rate	Error/Day	code errors detection rate	Error/Day
rework design errors rate	Error/Day	rework code errors rate	Error/Day
design modification escape errors rate	Error/Day	code modification escape errors rate	Error/Day
code modification errors generation rate	Error/Day	testing correction rate	Error/Day

Tab. 6: Generování chyb - toky

Výstup

V této části modelu nás nejvíce zajímá množství chyb, které byly během procesu změny software zaznamenány. Hladina *Reworked design errors* představuje chyby, jež byly nalezeny a odstraněny během fáze modifikace návrhu a stejně tak hladina *Reworked code errors* s tím rozdílem, že představuje chyby nalezené a odstraněné během modifikace kódu. Dále můžeme sledovat počet chyb, které nebyly nalezeny ani v jedné modifikační fázi a jejich počet je zaznamenán v hladinách *Escaped code errors*, respektive *Errors corrected in test*, jež zároveň představuje celkový počet chyb odstraněných ve fázi testování. Hodnoty zmíněných dvou hladin by měly být sobě rovny. Za poslední výstupní prvek bychom mohli považovat proměnnou *Design errors density in code*, která reprezentuje chyby návrhu ovlivňující chybovost ve fázi kódování.

Grafická reprezentace této části modelu z obrázku Obr. 10 nám umožní lépe si představit a pochopit vazby mezi jednotlivými prvky, jež byly výše v této kapitole popsány.



Obr. 10: Generování chyb referenčního modelu

4.3.4 Vynaložené úsilí

Poslední část z celkových tří, na které jsme referenční model rozdělili, bude představovat úsilí vynaložené v průběhu procesu změny softwarového systému. V modelu můžeme mapovat úsilí, které bylo vynaloženo na každou z jednotlivých fází procesu, veškeré vynaložené úsilí z každé fáze takové fáze procesu je následně soustředěno do jediného toku, v němž je vše sečteno a akumulováno v hladině *Total effort*. Tato část je co do rozsahu nejmenší, ale ve srovnání s oběma předchozími poskytuje neméně důležité informace.

Vstupní parametry

Pro výpočet celkového úsilí jsou vstupními parametry všechny toky v modelu, jež představují činnost pracovníků v jednotlivých fázích procesu změny software. Pro stanovení úsilí v procesech odstraňování chyb během modifikace návrhu, modifikace kódu a testování jsou vstupními parametry následující toky *rework design errors rate*, *rework code errors rate*, *testing correction effort rate*, které jsou přepočítávány na hodnoty s měrnou jednotkou *Effort*, jelikož v části modelu označené generování chyb, je pracováno s měrnou jednotkou *Errors*. Odvození mezi jednotkami bude popsáno v samostatné kapitole. Tabulka Tab. 7 zobrazuje seznam všech vstupních parametrů v té části modelu.

Vstupní parametr	**Měrná jednotka**
Rework design errors productivity	Effort/Error
Rework code errors productivity	Effort/Error
Error correction productivity	Effort/Error

Tab. 7: Vynaložené úsilí – vstupní parametry

Hladiny

Nejdůležitější hladinou je *Total effort*, která akumuluje veškeré vynaložené úsilí v rámci celého procesu vývoje. Za zmínku stojí také tři další hladiny *Reworked design errors effort*, *Reworked code errors effort*, *Errors correction effort*, jež reprezentují úsilí vynaložení na odstranění chyb. Tabulka Tab. 8 zobrazuje seznam všech hladin v té části modelu.

Hladina	Měrná jednotka
Reworked design errors effort	Effort
Reworked code errors effort	Effort
Errors correction effort	Effort
Total effort	Effort

Tab. 8: Vynaložené úsilí – hladiny

Toky

Stěžejním tokem, který plní hladinu *Total effort*, je tok *total effort rate*. Jeho hodnota v každém kroku simulace je určena součtem všech hodnot toků základního řetězce (kromě toku *requirements generation rate*) a toků, jež generují vynaložené úsilí na odstranění chyb. Tabulka Tab. 9 zobrazuje seznam všech toků v té části modelu.

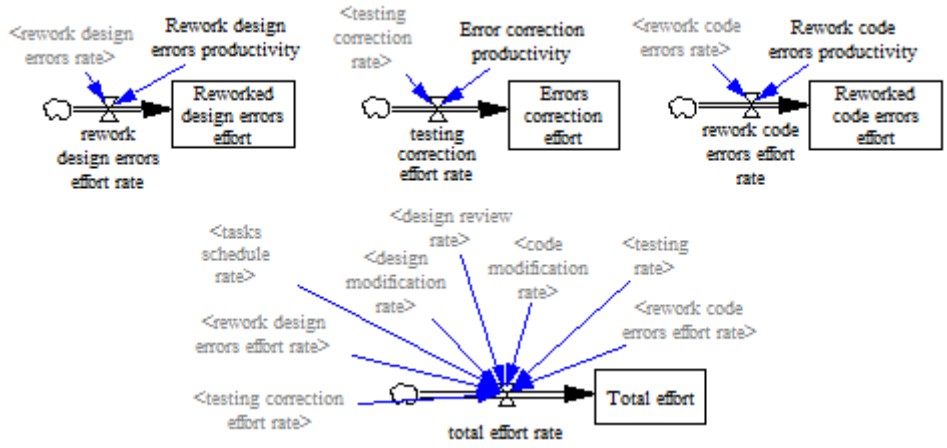
Tok	Měrná jednotka
rework design errors effort rate	Effort/Day
rework code errors effort rate	Effort/Day
testing correction effort rate	Effort/Day
total effort rate	Effort/Day

Tab. 9: Vynaložené úsilí – toky

Výstup

V této části modelu je vhodné zaměřit se na dva nejzajímavější výstupy. Bezesporu nejdůležitějším výstupem je tok *total effort rate*, z něhož můžeme vyčíst aktuálně vynakládané úsilí ve zvoleném čase. Graf časového průběhu tohoto toku je charakteristický tím, že na začátku rychle roste, jelikož se do procesu změny software postupně spustí všechny fáze, a poté postupně klesá, jakmile jsou některé fáze dokončeny. Druhým zajímavým výstupem, o jehož důležitost není pochyb, je hladina *Total effort*, jež udává celkové úsilí vynaložené v procesu změny software, což tvůrce modelu bude nepochybně při simulace zajímat.

Na obrázku Obr. 1, je grafické znázornění poslední části modelu. Toky, které se nacházejí v ostatních částech modelu a jsou vstupními parametry prvků v této části, jsou zobrazeny jako tzv. stínové proměnné (shadow variables).



Obr. 11: Vynaložené úsilí referenčního modelu

4.3.5 Převody měrných jednotek a vztahy mezi nimi

Výše v textu bylo na několika místech zmíněno označování měrných jednotek hladin *Requirements*, *Tasks*, *Errors* a *Effort*. V softwarovém inženýrství bývá zvykem uvádět rozsah software v jednotkách *SLOC* (Source Lines Of Code).

Při každém vytváření modelu je pro lepší orientaci vhodné zvolit si jednu či více základních jednotek, které budou vůči ostatním v postavení jednotek metrických, a vůči nim určit jednotlivé převodní vztahy. V našem referenčním modelu budou v části základního řetězce procesu změny software zvoleny coby metrické jednotky *Requirements* a *Tasks*. K jednotlivým převodním vztahům byly použity teze, které již ve své disertační práci vytvořil Raymond. J. Madachy. Pro účely našeho modelu budeme tedy vycházet z následujících předpokladů[31]:

- 1 Requirement odpovídá 1 Task
- 1 Task odpovídá 60 SLOC
- 1 SLOC odpovídá 1 Error
- Na 1 Task připadá 2,5% Error

Při procesu simulace generování chyb v tomto modelu by však docházelo k výsledkům velmi nízké hodnoty (v řádech desetinných čísel), a proto je vhodné přepočítat jednotky *Tasks* na *SLOC*. Výslednou hodnotou, která z této úpravy vyplyne, je poměr:

- na 1 *Task* připadá 1,5 chybných *SLOC*.

[31] MADACHY, R. *A Software Project Dynamics Model For Process Cost, Schedule And Risk Assessment*. California: 1994. 125 s. Disertační práce na Faculty Of The Graduate School University Of Southern California. Vedoucí disertační práce Dr. Barry Boehm, Dr. Behrokh Khoshnevis, Dr. Eberhardt Rechtin. s. 44-51.

Jak bylo výše podotknuto, jednou z nejdůležitějších hladin, jíž je třeba věnovat patřičnou pozornost, je celkové vyvinuté úsilí *Total effort*, které je součtem osmi toků (viz kapitola 4.3.4). Proto je nezbytné uvědomit si základní vztah mezi jednotkami *Task* a *Error*, se kterými tyto toky pracují. Hodnoty v tocích *tasks schedule rate*, *design modification rate*, *design review rate*, *code modification rate*, *testing rate* jsou vyjádřeny v jednotce *Task* – jedná se o množství *Tasks* zpracovaných za zvolenou časovou jednotku. Tyto toky jsou proto rovnou propojeny s tokem *total effort rate*, jenž generuje výsledné úsilí. Naproti tomu zbylé tři toky *rework design errors rate*, *rework code errors rate*, *testing correction rate* pracují s hodnotami vyjádřenými jednotkou *Errors*. Pro získání relevantní hodnoty celkového vyvinutého úsilí je tedy u těchto tří toků potřebné provést převod jednotek z *Error* na *Task*, a to v poměru:

- úsilí vyvinuté na 60 *Errors* odpovídá úsilí vyvinutému na 1 *Task*.

Převod je proveden odvozenými toky, které jsou pak ve výsledku napojeny na tok *total effort rate*.

4.3.6 Simulace modelu v nástroji Vensim

Cílem simulace ve zvoleném nástroji Vensim je vytvoření komplexního referenčního modelu, a to nejen z hlediska grafického, ale především z hlediska funkčního. Předběžnou grafickou podobu bylo možno již dříve vyvodit vzhledem k faktu, že zvolený referenční model představuje kompozici popisovaných částí základního řetězce, generování chyb a vynaloženého úsilí (viz Obr. 9, Obr. 10 a Obr. 11). Přesto je žádoucí shlédnout daný referenční model v kompletní podobě, což nám umožní konkrétněji vnímat vazby, vztahy a návaznosti mezi jednotlivými prvky modelu, se kterými jsme byli doposud seznámeni pouze v teoretické verbální formě. Grafická podoba modelu je vzhledem k jeho obsáhlosti znázorněna v samostatné příloze B.

V dosavadní deskripci problematiky simulace a modelování jsme se nezmínili o rovnicích, které jsou klíčovým artefaktem určujícím požadované chování celého modelu a také jsou jejich prostřednictvím počítány výstupní hodnoty simulací. Tvorba rovnic je, až na nečetné výjimky (rovnice generované v hladinách), záležitostí tvůrce modelu, jelikož odráží jeho konkrétní požadavky pro simulaci. Uveďme si krátký demonstrativní výčet rovnic, které mohou být při simulaci užity pro výpočet hladin, toků či proměnných. Kompletní seznam rovnic modelu je součástí přílohy C.

```
design modification rate = IF THEN ELSE(Scheduled tasks > 0, Design
      engineers*Design modification productivity, 0)

Design errors = INTEG( design modification errors generation rate-design
      errors detection rate-design modification escape errors rate, 0)

Design errors density in code = Escaped design errors/IF THEN
      ELSE(Cumulative modified design = 0, 1, Cumulative modified design)

Code modification errors density = 1.5

testing correction rate = Escaped code errors*testing rate*Error density
```

rework code errors effort rate = rework code errors rate/Rework code errors produktivity

Po sestavení sady rovnic je referenční model zcela připraven k provádění simulací. Konkrétní simulace je samozřejmě vždy odvislá od podoby rovnic, které tvůrce modelu pro daný případ sestavil.

V rámci edukace v tomto článku provedeme jednu z nesčetného množství simulací v referenčním modelu. Jelikož rozsah vytvořených rovnic kapacitně přesahuje možnosti tohoto článku, presumujme tyto rovnice a nazvěme naši simulaci, jež je na této presumpci rovnic založena, *Simulace x*. Níže v textu budou rozebrány některé klíčové hodnoty této *Simulace x*.

V tabulce Tab. 10 jsou zaznamenány výsledné hodnoty každého 25. kroku z celkových 350 kroků *Simulace x* referenčního modelu vztahující se k tokům *rework design errors rate*, *total effort rate* a k hladinám *Scheduled tasks, Errors corrected in test* a *Total effort*. Z hodnot zaznamenaných pro hladinu *Scheduled tasks* a toky *rework design errors rate* a *total effort rate* můžeme pozorovat, že nejdříve narůstají, jak tyto prvky zapojovány postupně do procesu změny software a nakonec hodnoty klesají, když skončí fáze, jichž jsou součástí. Naopak u hladin *Errors corrected in test* a *Total effort* si můžeme všimnout pouze růstového průběhu, jelikož tyto hladiny reprezentují celkový počet odstraněných chyb respektive celkové úsilí. Kompletní tabulka s hodnotami všech prvků referenčního modelu pro každý krok simulace je součástí přílohy A.

Time (Day)	Scheduled tasks	rework design errors rate	Errors corrected in test	total effort rate	Total effort
0	0	0	0	0	0
25	52	4,79812	28,3779	18,0687	415,331
50	102	4,8	82,0152	18,0716	867,099
75	152	4,8	137,013	18,0718	1318,89
100	112	4,8	192,113	12,0719	1680,69
125	12	4,8	247,241	12,0719	1982,49
150	0	0,00268102	296,499	3,97266	2106,55
175	0	3,60E-07	339,995	3,97203	2205,85
200	0	4,82E-11	383,207	3,972	2305,15
225	0	6,47E-15	419,517	1,50802	2377,56
250	0	8,67E-19	423,364	1,50033	2415,12
275	0	1,16E-22	423,521	1,50001	2452,62
300	0	1,56E-26	423,527	1,5	2490,12
325	0	2,09E-30	423,528	1,5	2527,62
350	0	2,80E-34	423,528	4,67E-36	2553,12

Tab. 10: Výsledné hodnoty Simulace x provedené v nástroji Vensim

5 Implementace systémové dynamiky v Javě

5.1 Tvorba simulačního engine

Druhou praktickou součástí této práce, kromě vytvoření funkčního referenčního modelu simulace vývoje změny softwarového systému za použití metody systémové dynamiky, bylo nakódování simulačního engine v jazyce Java. Pomocí tohoto enginu mělo dojít k simulaci modelů systémové dynamiky, měl se tedy stát paralelou k simulačnímu nástroji Vensim či k dalším obdobným nástrojům, jejichž demonstrativní výčet byl uveden výše v textu. Správnou funkčnost nově vytvořeného engine měla být prokázána porovnáním výsledků simulací s výsledky v nástroji Vensim. Soulad obou výsledků by měl být dostatečným důkazem ověřujícím správné fungování vytvořeného engine, a to zejména vzhledem k faktu, že se jedná o simulační nástroj s několikaletou praxí vytvořený na profesionální úrovni, tudíž by jeho výsledky měly být brány jakožto neoddiskutovatelné.

Následně došlo k propojení vzniklého simulačního engine s uživatelským rozhraním, jehož vytvoření bylo předmětem práce jiného autora. Okrajové pojednání o tomto uživatelském rozhraní bude pro lepší pochopení komplexnosti, propojenosti a významu prací obou autorů uvedeno i v části této kapitoly.

5.2 Použité technologie

5.2.1 Java

Java je moderní objektově orientovaný programovací jazyk a platforma, jež byly vyvinuty a v roce 1995 představeny společností Sun Microsystems, která byla později připojena ke společnosti Oracle. V roce 2007 Sun Microsystems uvolnila zdrojové kódy většiny Java technologií pod licencí GNU, což znamenalo, že byla od té doby Java dále vyvíjena jako open source produkt. Nejen toto zpřístupnění, ale i kvalita a širokospektrálnost jejího provedení vedlo k tomu, že došlo k masivnímu rozšíření Javy a k dnešnímu datu využívá tuto platformu více než 6,5 milionů softwarových vývojářů po celém světě. Zasahuje snad do všech oblastí průmyslu, výroby a vývoje a má zastoupení v široké řadě nejrůznějších zařízení, počítačů, systémů a sítí.[32]

Jednou z nejvýraznějších charakteristik Javy je její přenositelnost, což v praxi znamená, že aplikace nakódované v tomto jazyce mohou být spouštěny na libovolném hardware či operačním systému. Tato vlastnost je zajištěna překladem Java zdrojového kódu na tzv. mezikód (bytecode),

[32] *Java* [online]. [cit. 2012-04-28]. Dostupné z: < http://java.com/en/about/>.

který je na konkrétním zařízení interpretován, či případně za běhu přeložen do nativního kódu. To je pochopitelné vlastností znamenající obrovské ulehčení a variabilitu použití. Jelikož je Java jazykem rozšířeným a mezi širokou odbornou veřejností důvěrně známým, nepokládáme za nutné rozvádět další její charakteristické vlastnosti, a proto ty nejvýznamnější uvedeme k heslovitém výčtu. Jedná se kupříkladu o generační správu paměti, jež je realizována pomocí automatického Garbage collectoru, důraz kladený na bezpečnost pro ochranu počítače v síťovém prostředí, před nebezpečnými operacemi nebo napadením jiným škodlivým kódem, dále Java obsahuje mechanismus vláken umožňující spouštět více úloh najednou, a mnoho dalších vlastností, jejichž popis by svou kvantitou mohl odpovídat samostatné publikaci.

Pro implementaci simulačního engine vyhodnocující modely systémové dynamiky byla v této práci použita Java platforma Java Platform, Standard Edition 6 Development Kit (JDK). JDK je vývojové prostředí zahrnující také JRE potřebné ke spouštění aplikací.

5.2.2 Vývojové prostředí Eclipse

Eclipse bychom mohli chápat jako open source komunitu, jejíž projekty jsou zaměřeny na budování otevřené vývojové platformy složené z frameworku podporujícího pluginy, nástrojů a prostředí ke spouštění aplikací (tzv. runtime) určených pro vývoj, budování a správu software napříč všemi fázemi jeho životního cyklu.[33]

Eclipse Project původně vyvinula společnost IBM na konci roku 2001a jeho podporu zajišťovalo konsorcium různých dodavatelů software. Začátkem roku 2004 byla založena nezávislá nezisková organizace Eclipse Foundation, jež měla zaštítit a spravovat Eclipse komunitu. V dnešní době je komunita složena z jednotlivců a organizací napříč celým širokým spektrem softwarového průmyslu. Od roku 2006 pak Eclipse Foundation pravidelně vydává novou verzi, která vždy zahrnuje platformu Eclipse a také mnoho dalších Eclipse projektů. Doposud poslední verze této platformy s číslem 3.7 nesoucí označení Indigo, jež je použita také v této práci, byla vydaná v polovině roku 2011.[34]

Samotný Eclipse v základní verzi obsahuje pouze Eclipse JDT nabízející IDE s vestavěným Java kompilátorem a kompletním modelem javovských zdrojových souborů. Většina lidí tak Eclipse považuje za vývojové prostředí pro tvorbu aplikací v jazyce Java. Ovšem díky mechanismu, jenž umožňuje rozšiřitelnost pomocí pluginů, je možné zakomponovat podporu mnoha dalších programovacích jazyků (např. C++, PHP, Phyton aj.) či jiných nástrojů umožňujících kupříkladu vizuální návrh grafických uživatelských rozhraní desktopových aplikací, modelování UML diagramů či zápis do HTML.[35]

---

[33] *Eclipse* [online]. [cit. 2012-04-28]. Dostupné z: <http://www.eclipse.org/org/>.
[34] Tamtéž
[35] *Eclipse (vývojové prostředí).* Wikipedie [online]. Změněno 6.3.2012 [cit. 2012-04-28]. Dostupné z: <http://cs.wikipedia.org/wiki/Eclipse_%28v%C3%BDvojov%C3%A9_prost%C5%99ed%C3%AD%29>.

5.2.3 JUnit

JUnit je framework, který byl navržen pro vytváření spouštění jednotkových testů v Javě. Díky přívětivému uživatelskému rozhraní umožňuje vývojáři pohodlně a rychle vytvářet testovací případy pro ověření správnosti každého pokroku a odhalení chyb při vývoji aplikace. Spouštění testů je velmi rychlé a je možné je nepřetržitě opakovat. Po dokončení všech testovacích případů je vývojář ihned informován o výsledcích a dozví se tak téměř okamžitě, které testovací případy byly úspěšné a které naopak selhaly. Díky tomu může vývojář velmi rychle opravit vzniklé chyby již během vývoje aplikace.[36]

Unit testy byly použity také během vývoje simulačního engine v rámci této práce, a to konkrétně k verifikaci výsledných hodnot simulovaných modelů. K těmto účelům bylo použito frameworku JUnit ve verzi 4.10.

5.2.4 Apache Maven

Maven je multiplatformní nástroj pro správu, řízení a automatizaci buildů aplikací, jenž pochází z dílny Apache Software Foundation a je vydáván pod licencí Apache Licence 2.0[37]. Počátkem roku 2012 byla vydána doposud poslední verze s označením 3.0.4, jež byla použita pro sestavení výsledného projektu v této práci.

Maven je vývojové prostředí, které nedisponuje žádným grafickým uživatelským rozhraním, ale je ovládáno pouze pomocí příkazových řádků. Maven je svou funkcionalitou velmi podobný nástroji Ant, avšak poskytuje mnohem snadnější konfiguraci a nabízí tak širší možnosti při sestavování projektů, a to díky modulární architektuře fungující na principu volání jednotlivých pluginů. Všechny pluginy jsou volány jednoduchým příkazem `mvn [nazev_pluginu]:[goal]`, kde *goal* označuje funkci, jež je volána.

Hlavním cílem Mavenu je umožnit vývojářům získat přehled o stavu projektu v co nejkratším možném čase. Za účelem dosažení tohoto cíle určili jeho tvůrci několik oblastí zájmů, které by měl Maven řešit. Jsou jimi:

- usnadnění procesu sestavení – ačkoliv se při používání uživatel nezbaví nutnosti znát základní mechanismy potřebné k sestavení, Maven přesto poměrně značně odstiňuje potřebu znát jejich detaily, tudíž celý proces výrazně zjednodušuje
- zajištění jednotných sestavovacích systémů – sestavování projektu probíhá pomocí POM (Project Object Model) a množiny pluginů, které jsou sdíleny všemi projekty používajícími Maven

---

[36] *JUnit.* Search software Quality [online]. [cit. 2012-04-30]. Dostupné z: <http://searchsoftwarequality.techtarget.com/definition/JUnit >.
[37] *Project License.* Apache Maven Project [online]. [cit. 2012-04-30]. Dostupné z: <http://maven.apache.org/license.html>.

- zajištění kvalitních informací o projektu – Maven poskytuje mnoho užitečných informací o projektu, jež jsou částečně převzaty z POM a částečně jsou generovány ze zdrojů projektu. Jako demonstrativní výčet uveďme informace o změnách v projektu, křížené odkazování zdrojů, seznam závislostí či reporty z jednotkových testů a informace o jejich pokrytí
- poskytnutí osvědčených postupů při vývoji – Maven sdružuje základní principy osvědčených postupů uplatňovaných při vývoji tak, aby bylo snadné v tomto směru projekt řídit. Dále také poskytuje určitá doporučení, jak uspořádat adresářovou strukturu, která bude umožňovat snadnější orientaci v projektu
- poskytnutí transparentního přidávání nových funkcí – pro uživatele používající Maven je poskytován snadný způsob, jak jeho instalaci aktualizovat, a to díky jeho modulární architektuře[38]

Adresářová struktura

Jednou z nejdůležitějších zásad Maven projektu je, že všechny v něm zakomponované soubory mají přesně definované umístění. Používání předepsané struktury adresářů je doporučováno, avšak není to nutností. To s sebou přináší tu výhodu, že Maven projekty mají identickou strukturu adresářů, a proto je snadnější se v ní orientovat. Pokud není možné tuto strukturu dodržet, lze ji změnit pomocí POM.

`Kořenový adresář` – obsahuje pom.xml a ostatní adresáře
`src/main/java` – obsahuje kompilovatelné .java soubory
`src/main/resources` – obsahuje další soubory např. konfigurační XML
`src/test/java` – obsahuje třídy pro unit testy
`src/test/resources` – obsahuje konfigurační soubory pro unit testy

POM

Maven ke své činnosti používá model Project Object Model, jenž je v praxi známější pod svou zkratkou POM. Tento model je reprezentován jednoduchou XML strukturou s názvem pom.xml, ve které jsou definovány jednotlivé části projektu, používané pluginy a taktéž jeho závislosti na externích knihovnách a nástrojích. Pokud nastane situace, kdy je konkrétní projekt složen z více dílčích projektů, naskýtá se možnost vytvořit jeden nadřazený POM, ze kterého budou ostatní posléze dědit jeho vlastnosti. Díky tomuto se dochází k obrovskému usnadnění, jelikož lze sestavit celý projekt jediným příkazem.

[38] *Introduction.* Apache Maven Project [online]. [cit. 2012-04-30]. Dostupné z: <http://maven.apache.org/what-is-maven.html>.

Eclipse plugin m2eclipse

Apache Maven lze integrovat přímo do vývojového prostředí Eclipse pomocí pluginu s názvem m2eclipse.Tento plugin přináší řadu užitečných funkcí, kupříkladu spouštění sestavování Maven projektů přímo z Eclipse, řízení závislostí cesty pro sestavování projektu mezi Eclipse a Maven pom.xml, automatické stahování požadovaných závislostí ze vzdálených Maven repositářů, snadnější editaci pom.xml aj. Vývojáři v jazyce Java tak určitě ocení velmi těsnou integraci s JDT, jež výrazně zjednodušuje užívání Java artefaktů.

5.3 Knihovna JEval

JEval je Java knihovna pracující s matematickými, stringovými, booleanovskými a funkcionálními výrazy, které dokáže parsovat a ohodnocovat. Je v ní zahrnuto 39 nejrůznějších matematických a stringových funkcí, podporuje všechny matematické a booleanovské operátory a v neposlední řadě také práci s proměnnými a jejich řešení. Jako demonstrativní výčet uveďme některé z podporovaných operátorů a funkcí:

- Operátory: (,), +, -, *, /, %, =, !=, <, <=, >, >=, &&, ||, !
- Matematické funkce: ln, exp, abs, cos, sin, pow, sqrt, min, max, round aj.
- Stringové funkce: charAt, compareTo, concat, equals, replace, trim aj.

Kromě široké škály využití, které tato knihovna nabízí při řešení různých projektů pracujících s matematickými funkcemi či textem, je její nespornou výhodou je bezplatné poskytnutí k volnému použití. Knihovna JEval je pro tuto práci použita v její poslední verzi 0.9.4 Beta z prosince 2008[39].

5.3.1 Použití při implementaci

Hlavním vstupním bodem k celému API knihovny JEval je třída `Evaluator`, která disponuje všemi potřebnými metodami pro práci s matematickými, stringovými, booleanovskými či vlastnoručně definovanými funkcemi. Pro naši potřebu jsou při implementaci nejdůležitější metody `replaceVariables` a `evaluate`. První ze zmíněných slouží k nahrazování proměnných v rovnicích jejich hodnotou (číslem nebo opět rovnicí). Použití této metody je nezbytné, jelikož v systémové dynamice se nezřídka stává, že výpočet proměnné je zanořený hlouběji ve struktuře modelu, s čímž si později využitá metoda `evaluate` neporadí, vzhledem k faktu, že pracuje pouze s proměnnými v rámci jedné úrovně. Proto je nejprve potřeba vše spojit pomocí `replaceVariables` do pomyslné jedné úrovně a takto vytvořené rovnice poté lehce vypočítat s přispěním metody `evaluate`. Z třídy `Evaluator` využijeme pak ještě další dvě metody `setVariables` a `clearVariables`. Zatímco první z nich nastaví vstupní proměnné a jejich hodnoty, jež jsou definovány v simulovaném modelu, druhá metoda tyto proměnné odstraní.

[39] JEval [online]. [cit. 2012-04-30]. Dostupné z: <http://jeval.sourceforge.net/>.

5.3.2 Omezení a nevýhody

Je nutné připustit, že má tato knihovna také jisté nevýhody. Jejich existence však při použití knihovny nepůsobí větší komplikace. Jednou z nejmarkantnějších nevýhod je fakt, že všechny proměnné, které jsou v modelu definovány, je potřeba zapisovat ve speciální notaci `#{nazev_promenne}`, jelikož pouze v tomto tvaru knihovna jednotlivé proměnné rozeznává. Z toho vyplývá, že tuto notaci `#{` nelze použít ve výrazech a rovnicích. Další zjištěnou nevýhodou je nutnost uzavírání všech matematických výrazů do závorek, aby docházelo ke správnému vyhodnocování rovnic. Kupříkladu namísto výrazu `A+B` je žádoucí provést zápis jako `(A+B)`. Takovýto zápis je důležitý ve fázi ohodnocování proměnných, kde může nastat situace, že je za určitou proměnnou dosazena rovnice. Bez závorkového zápisu by se totiž mohlo stát, že nebude dodrženo správné pořadí matematických operací a výsledky nebudou korektní. Na malém příkladu demonstrujme situaci, která může nastat:

$$X = Y * Z$$
$$Y = A + B$$
$$Z = C - D$$
$$X = A + B * C - D$$

Proměnné X, Y, Z považujme za prvky simulovaného modelu. Z příkladu je patrné, že jakmile budou metodou `replaceVariables` do první rovnice X dosazeny za proměnné příslušné výrazy, dojde k situaci, kdy bude násobení upřednostněno před sčítáním a odčítáním, což by byl matematicky korektní postup, ale nelze ho takto uplatnit pro naše účely. Výsledná hodnota rovnice X bude tedy nesprávná, jelikož by měly být nejdříve vypočítány rovnice Y a Z a až poté jejich výsledné hodnoty dosazeny do rovnice X. Pro dosažení správných výsledných hodnot všech prvků při simulaci musíme takovýto postup výpočtu dodržovat.

5.4 System dynamics laboratory[40]

System dynamics laboratory je uživatelské rozhraní (GUI) umožňující tvorbu a simulaci modelů systémové dynamiky. Jeho vývoj probíhal paralelně s vývojem v této práci popisovaného simulačního engine, který byl poté do této aplikace integrován pro vykonávání výpočtů v simulovaném modelu.

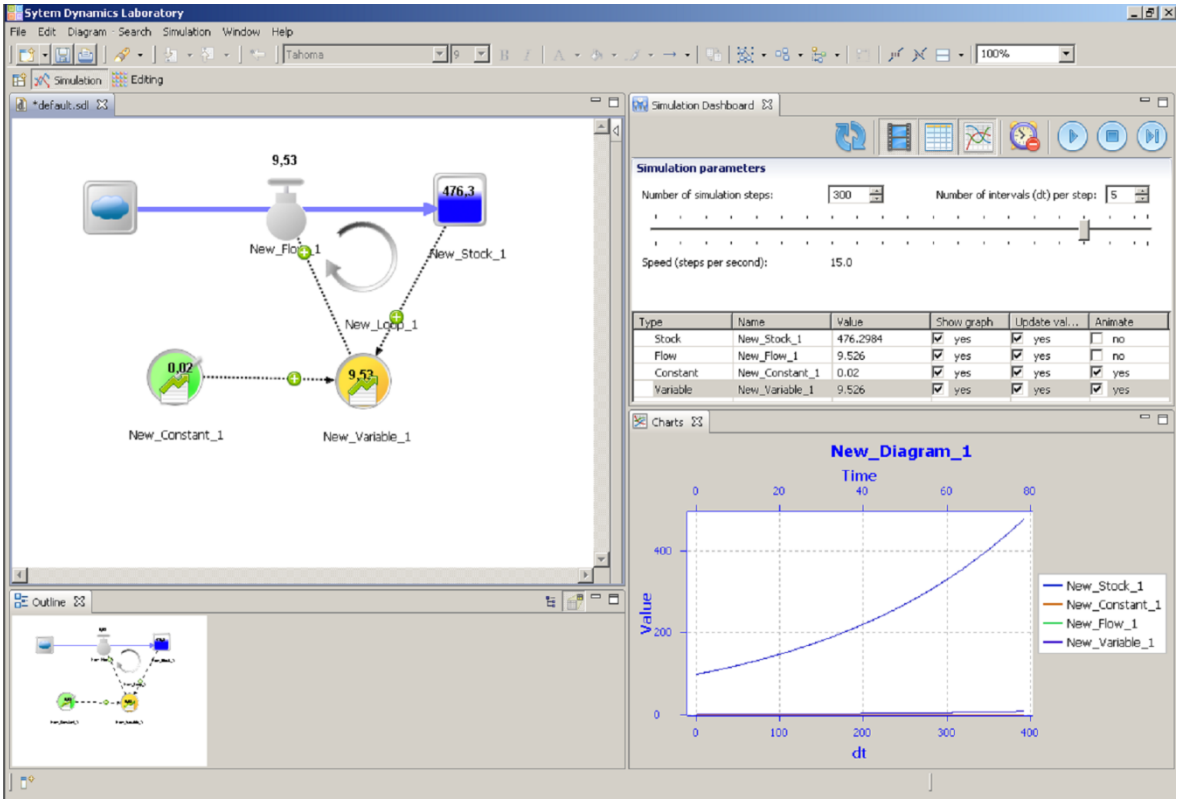
Aplikace uživatelského rozhraní je postavena na Eclipse RCP frameworku, jehož koncepce je postavena na pluginech. Plugin je zde základní jednotkou, ze které je celá platforma poskládána a pomocí které ji lze dále rozšiřovat. Pro správu všech pluginů, jejich vzájemnou komunikaci a řízení jejich životního cyklu je využíván framework OSGi. Jedná se o samostatnou technologii, která je na RCP nezávislá a je primárně zaměřená právě na poskytování modularity tvůrcům Java aplikací. Díky tomu je výsledná aplikace velmi jednoduše rozšiřitelná a lze ji upravovat bez zásahů

[40] Použito z konceptu diplomové práce: Melcr, Jiří. Simulace Systémové dynamiky v prostředí Eclipse RCP, 2012.

do zdrojových kódů. Také náš simulační engine byl k System dynamics laboratory připojen jako jeden z pluginů.

Za nejdůležitější část celé aplikace System dynamics laboratory bychom mohli považovat nástroj vytváření interaktivních diagramů, které graficky reprezentují modely systémové dynamiky. Pro tvorbu tohoto nástroje využil autor jeden z projektů Eclipse komunity, který je znám pod označením GMP (Graphical Modeling Project). Ten volně sdružuje několik dalších projektů, jejichž společným jmenovatelem je vývoj a využití grafických modelovacích nástrojů. Jedním z nich, jenž byl použit pro vývoj tohoto modelovacího nástroje, je EMF (Eclipse Modeling Framework). Ten by mohl být stručně popsán jako prostředek pro generování kódu, s jehož pomocí lze vytvářet nástroje a aplikace na základě strukturovaných datových modelů. Dalším z projektů GMP je GEF (Graphical Editing Framework), který poskytuje o stupeň vyšší funkcionalitu než zmiňovaný EMF. Umožňuje manipulaci s tzv. metamodelem vytvořeného na základě modelu EMF formou grafického editoru. Posledním projektem, který byl pro účely modelovacího nástroje použit, je GMF (Graphical Modeling Framework). Ten umožňuje pomocí šablon vygenerovaný grafický editor nakonfigurovat tak, aby jeho jednotlivé části odpovídaly příslušným částem doménového modelu.

Výsledné uživatelské rozhraní této aplikace poskytuje všechny důležité funkce, které jsou pro vytváření a simulování modelů systémové dynamiky důležité. V našem případě nás budou nejvíce zajímat ovládací prvky, jejichž funkčnost je součástí implementace v této práci. Na obrázku Obr. 12 je zobrazena výsledná podoba nástroje System dynamics laboratory.



Obr. 12: Ukázka simulačního nástroje System dynamics laboratory

5.5 Architektura

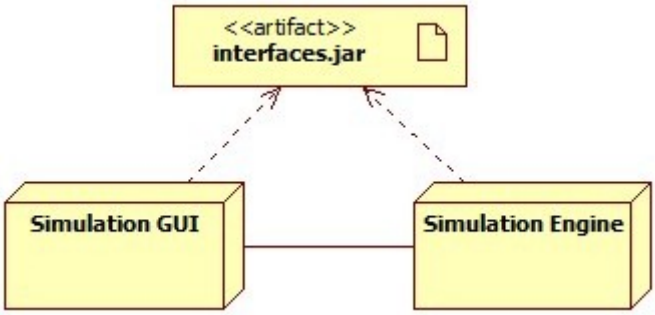
Konečná podoba architektury knihovny obsahující simulační engine je do značné míry přizpůsobená uživatelskému rozhraní, jež umožňuje vizualizovat a spouštět modely systémové dynamiky. Spojení těchto dvou nezávislých projektů přineslo nesnáze v podobě nalezání cesty pro vzájemnou komunikaci mezi simulačním enginem a uživatelským rozhraním (GUI), jelikož myšlenka vzájemné integrace vznikla až v pozdější fázi vývoje obou zprvu samostatných částí, čímž došlo k celkovému ovlivnění výsledné podoby implementační části této práce. Přes tyto prvotní nesrovnalosti se ve výsledku podařilo obě části propojit a vytvořit tak funkční a plně použitelný celek simulačního nástroje.

5.5.1 Propojení simulačního engine a GUI

Pro vzájemnou komunikaci mezi simulačním enginem a GUI byla vytvořena knihovna `Interfaces.jar` obsahující rozhraní, která musejí implementovat obě části celkového simulačního nástroje. Tato rozhraní deklarují třídy a jejich metody či datové struktury pro ukládání hodnot, pomocí nichž je možné z GUI posílat do simulačního enginu vstupní hodnoty, ovládat průběh simulace a naopak ze simulačního engine pak vrátit výsledné hodnoty. Pro tyto účely jsou v knihovně `Interfaces.jar` zahrnuta tato rozhraní:

- `ISimulationElement` – představuje prvek v modelu a uchovává jeho jméno, rovnici, počáteční hodnotu a aktuální hodnotu v průběhu simulace
- `ISimulationEngine` – zajišťuje nastavení počátečních parametrů simulace, řízení jejího průběhu a předávání výsledných hodnot simulovaného modelu
- `ISimulationListener` – popisuje odběratele průběžných výsledků simulovaného modelu
- `ISimulationRequest` – pomocí datových struktur popisuje prvky diagramu simulovaného modelu, které jsou takto předávány simulačnímu engine
- `ISimulationResponse` – popisuje způsob, kterým jsou vráceny prvky diagramu simulovaného modelu, uchovávající původní instance objektů, jež byly předány v rozhraní ISimulationRequest

Na závěr tohoto pojednání se ještě krátce zmiňme o důležitosti rozhraní `ISimulationRequest` a `ISimulationResponse`, jež slouží k předávání prvků diagramu simulovaného modelu mezi GUI a simulačním enginem. Tyto prvky jsou reprezentovány objekty implementujícími rozhraní `ISimulationElement` a pro GUI je důležité, aby po provedení simulace obdržel od simulačního engine reference na stejné instance, které mu před tím poskytl. Bližší specifikace výše uvedených rozhraní popisuje příslušná dokumentace knihovny `Interfaces.jar`. Na následujícím obrázku Obr. 13 je pak graficky znázorněno propojení GUI a simulačního engine.



Obr. 13: Diagram propojení simulačního engine a GUI

5.5.2 Architektura simulačního engine

Jak již bylo zmíněno výše v textu, architektura simulačního engine byla do značné míry přizpůsobena GUI, které tento engine využívá pro své účely. I přesto je tato architektura navržena tak, aby bylo možné simulační engine využít také v jiných GUI, než v tomto konkrétním, s nímž byl primárně v této práci propojen.

Jednou z nejdůležitějších tříd v rámci celé architektury je třída `Simulation`, která umožňuje GUI komunikovat s celým simulačním engine. Pro tyto účely implementuje rozhraní `ISimulationEngine`, které definuje metody pro nastavení parametrů simulace a ovládání průběhu simulace. Nejprve je volána metoda `setUp()` s parametry, jež obsahují vstupní diagram simulovaného modelu reprezentovaného instancí třídy implementující rozhraní `ISimulationRequest` a dále pak parametry pro nastavení celkového počtu kroků, velikost kroku dt a pausu mezi jednotlivými kroky v milisekundách. Nakonec je volána metoda `init()`, která nastaví potřebné parametry simulačnímu vláknu a přenese vstupní objekty reprezentující diagram simulovaného modelu do objektů používaných pro interní výpočty v třídě `SimulationEngine`. Jejich struktura bude popsána později v textu. V metodě `init()` je také předáván seznam všech registrovaných posluchačů, kteří jsou odběrateli výsledných hodnot všech prvků simulovaného modelu. Bezparametrickou metodou `run()` je poté simulace spuštěna. Uživatel je posléze postaven před volbu, jakým způsobem bude celou simulaci řídit. Výsledné hodnoty může dostat buď najednou, nebo je může dostávat ve zvolených časových intervalech. Tyto intervaly lze měnit i za běhu simulace pomocí metody `setTimeout()`, která má jako parametr novou hodnotu časové prodlevy mezi jednotlivými výsledky. Uživatel má možnost dále simulaci pozastavit a opětovně spustit, což je zajišťováno pomocí metody `pause()`. První zavolání této metody simulaci pozastaví a její opakované zavolání simulaci zase spustí. Pokud chce uživatel sledovat simulaci po jednotlivých krocích, má možnost ručně průběh simulace krokovat pomocí metody `step()`. Pro ukončení simulace pak slouží metoda `stop()`. Tato třída obsahuje také metody registrSimulationListener() pro registrování posluchačů a removeSimulationListener() sloužící k jejich odregistrování.

Druhou nejdůležitější třídou je `SimulationEngine`. Ta by se dala považovat za výpočetní jádro celého simulačního engine, jelikož obsahuje metody, které provedou výpočty rovnic všech vstupních prvků simulovaného modelu a poskytnou GUI výsledné hodnoty těchto prvků. Třída dědí z třídy `Thread`, která definuje základní metody pro spuštění, zastavení a ukončení vlákna. Vlákno je spouštěno metodou `run()`, která kontroluje stav simulace. Pokud je ukončena notifikuje o tom všechny posluchače v opačném případě, jestliže zbývá provést další kroky, volá metodu `tick()`. Poté je vlákno uspáno na dobu odpovídající hodnotě nastavené buď při volání metody `setUp()` nebo během simulace metodou `setTimeout()`. Po skončení každého kroku jsou opět všichni posluchači notifikováni a jsou jim předány výsledné hodnoty všech prvků modelu. Implementaci metody `run()`, nám přiblíží následná ukázka zdrojového kódu:

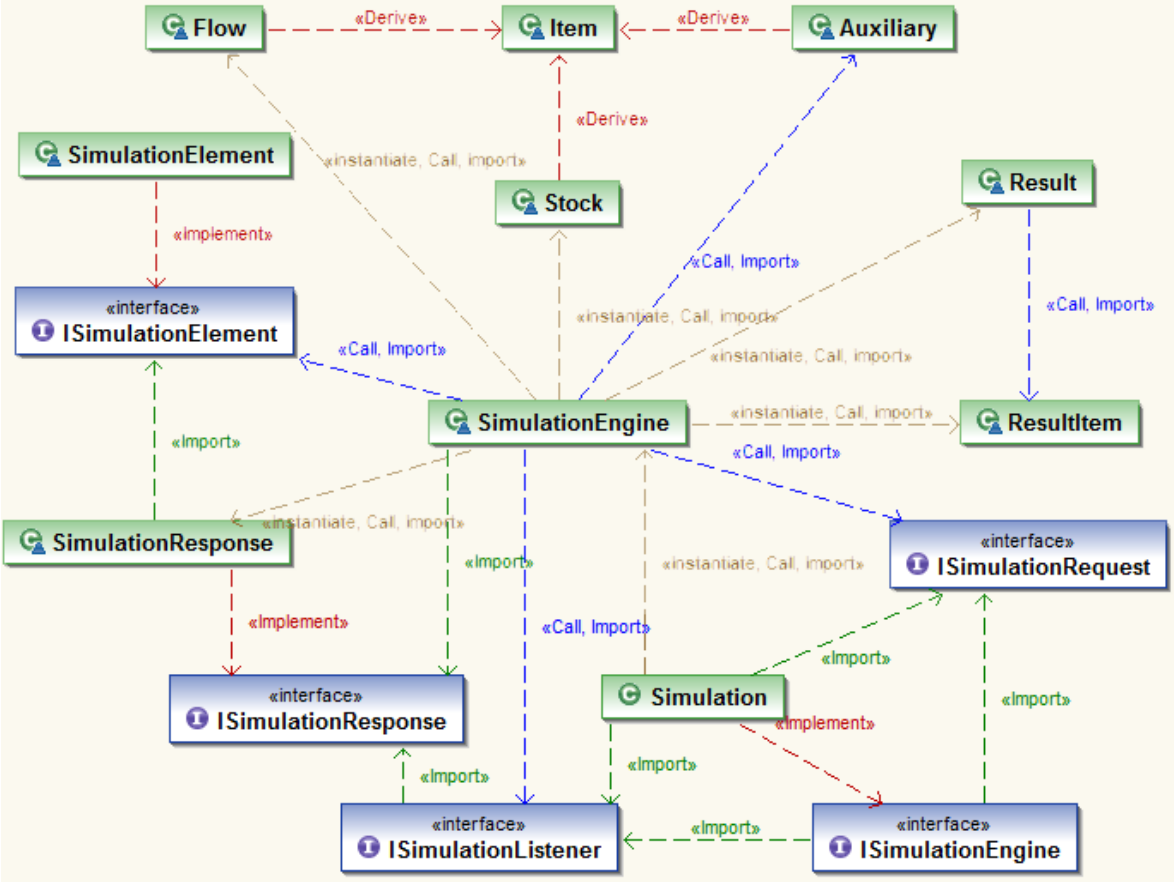
```
@Override
public void run() {
	try {
		for (this.positionInSimulation = this.startSimulation;
		this.positionInSimulation < this.endSimulation;
		this.positionInSimulation = this.positionInSimulation +
		(1 / this.stepSimulation)) {
			if (this.end) {
			notifyAllListenersEnd();
			return;
			}
			while (this.pause) {
				Thread.sleep(1000);
			}
			tick();
			Thread.sleep(getTimeOfStepSimulation());
		}
		notifyAllListenersEnd();
	}
	catch (final InterruptedException e) {
		notifyAllListenersAboutError(e.getMessage());
	}
}
```

Podobně jako `run()` funguje také metoda `nextStep()`, která taktéž volá metodu `tick()`, avšak s tím rozdílem, že ji zavolá pouze jednou a o jeden krok posune aktuální pozici v simulaci. Tím je zajištěno, že pokud další volání metody `tick()` bude pokračovat krokem, jenž následuje za tím předchozím. Synchronizovaná metoda `tick()` posune simulaci o jeden krok a zavolá metodu `move()`, která vypočítá výsledné hodnoty prvků modelu, dle jejich vstupních hodnot a rovnic. Bližší popis této metody je uveden v kapitole 5.5.3.

Součástí architektury simulačního engine jsou také třídy Auxiliary, Flow, Stock a jejich rodičovská třída Item, jež jsou používány pro interní výpočty v třídě SimulationEngine. Tyto třídy slouží k reprezentaci diagramu simulovaného modelu a jsou na ně mapovány objekty prvků, jež

byly do simulačního engine poslány z GUI. V tomto případě bychom mohli vynechat třídu Auxiliary, která má stejné vlastnosti jako třída Flow, ale byla ponechána pro přehlednost a lepší orientaci ve zdrojovém kódu. Třídy Result a ResultItem uchovávají hodnoty coby výsledky každého jednotlivého kroku simulace, jež jsou vždy k dispozici pro každý prvek modelu. Pro GUI je pak třída Result mapována na třídu SeimulationRespons, v níž jsou zpětně vráceny reference na objekty, které byly simulačnímu engine předány jako vstup.

Obrázek Obr. 14 nám umožňuje lépe si představit všechny vztahy mezi výše popsanými třídami a rozhraními.



Obr. 14: Diagram tříd simulačního engine

5.5.3 Princip řešení

Nejdůležitější částí knihovny je oblast výpočtu rovnic, které jsou pro simulaci nezastupitelné a ze kterých se celý model systémové dynamiky de facto skládá. Pro výpočet rovnic existuje několik postupů, např. použití matematických knihoven. Naše řešení bude rovnice modelu počítat jakožto soustavu rovnic s využitím parsovacích funkcí. Zvolené řešení je založeno na principu nahrazení proměnných v rovnicích modelu jejich hodnotami, kterými mohou být číselné

konstanty či rovnice. V případě, kdy je dosazena za proměnnou rovnice, jsou její proměnné znovu nahrazovány, dokud nevznikne rovnice bez proměnných. Takto nově sestavené rovnice jsou vypočítány a jejich finální výsledky uloženy.

O výpočty rovnic se stará metoda `move()` v třídě `SimulationEngine`, která obdrží jako vstupní parametry aktuální krok, ve kterém se simulace nachází, a počet kroků, které mají být spočteny. Tím je zajištěna možnost okamžitého dokončení simulace v případě, že jsou vráceny výsledky nejprve po manuálním krokování či automaticky ve zvolených časových intervalech. Výpočty každého kroku jsou řízeny jedním hlavním cyklem, který se opakuje dle vstupních parametrů, jež byly předány metodě `move()`. Na začátku každého kroku je metodou `fillWithElementValue()` vytvořena jedna `HashMap` obsahující všechny rovnice prvků flow, auxiliary a akumulující se hodnoty stock. Takto vytvořená mapa je poté poslána jako parametr metody `setVariables()`, jenž nastaví všechny potřebné rovnice a hodnoty dříve vytvořené instanci třídy `Evaluator` z knihovny JEval. Výpočty výsledných hodnot všech prvků modelu v rámci jednoho kroku simulace bychom mohli rozdělit na dvě části. V první části jsou vypočítány výsledné hodnoty všech flow a auxiliary tak, že nejdříve jsou v rovnicích nahrazovány všechny proměnné metodou `replaceVariables()`, až vzniknou rovnice složené pouze z číselných hodnot. Takto vytvořené rovnice jsou vypočítány metodou `evaluate()` a jejich výsledné hodnoty uloženy do mapy obsahující průběžné výsledky. Ve druhé části výpočtu výsledných hodnot simulace jednoho kroku jsou stejným způsobem, jako flow a auxiliary, vypočítány akumulující se hodnoty prvku stock. Pokud je nastavena velikost dt (počet výpočtu v jednom kroku) na jinou hodnotu než jedna, pak se hodnoty stock mění o podílnou část přitékajících i odtékajících flow určenou velikostí dt. Nakonec výpočtu jednoho kroku jsou výsledky uloženy pomocí metody `fillResultMap()`, která pro každý krok uloží seznam obsahující instance třídy `ResulItem`. Všechny důležité části metody `move()`, jež byly popsány, zobrazuje následující výsek zdrojového kódu:

```
for (double step = from; step < howmuch; step = step + 1 /
this.stepSimulation) {
      fillWithElementValues();
      ...
      for (final String key : this.calculatedEquations.keySet()) {
            final String calculatedEquation =
            evaluator.replaceVariables(this.calculatedEquations.get(key))
            ;
            final String resultEquation =
            evaluator.evaluate(calculatedEquation);
            final Double result = new Double(resultEquation);
            final String formatedResult =
            preventingErrorsInExp(this.formatter.format(result));
            calculatedEquationsResults.put(key, formatedResult);
      }
      ...
      for (final String key : this.stocksEquations.keySet()) {
            final String calculatedEquation =
            evaluator.replaceVariables(this.stocksEquations.get(key));
```

```
                final String resultEquation =
                evaluator.evaluate(calculatedEquation);
                final String temp = this.stocksAcumulationResults.get(key);
                final Double a = new Double(resultEquation);
                final Double b = new Double(temp);
                final Double result = b + a / this.stepSimulation;
                final String formatedResult = this.formatter.format(result);
                this.stocksAcumulationResults.put(key, formatedResult);
        }
        fillResultMap(step, calculatedEquationsResults);
}
```

5.5.4 Úskalí během implementace

Vývoj a následná implementace každého software v určitých fázích doprovází chyby, problémy a úskalí, s nimiž se musí vývojář potýkat a řešit je. Také během této práce nastaly situace, kdy se zdály být zvolené postupy nesprávné a vyvíjená aplikace nesplňovala očekávané chování. Vždy bylo vyvinuto maximální úsilí na opravu takovýchto nedostatků s cílem přivést aplikaci do požadovaného stavu. Uveďme nyní některé problémy, jež při implementaci simulačního engine v této práci nastaly a způsoby řešení, pomocí kterých byly odstraněny.

Prvním nedostatkem, který byl odhalen již v raných stádiích vývoje simulačního engine, byl problém nesprávného vyhodnocování rovnice některých prvků simulovaného modelu. Tento problém byl již podrobněji popsán v kapitole 5.3.2. Řešení se naskytlo v použití závorkové notace, kdy je před každý výpočtem rovnice prvku uzavřena do kulatých závorek, čímž je dodrženo žádoucí pořadí výpočtu výsledných hodnot všech prvků simulovaného modelu. Tímto přístupem byla také odstraněna další problémová část, se kterou si evaluační metody knihovny JEval nedokázaly poradit. V případě, že má hladina pouze vypouštěcí tok a její rovnice je definována zápisem `-#{flow}`, může nastat situace, kdy je za proměnnou `#{flow}` dosazeno záporné číslo a vznikne tak např. zápis `--10`. S tímto případem si metoda `evaluate` nedokázala poradit a vypsala chybové hlášení o nesprávném zápisu zpracovávaného výrazu. Uzavřením výsledné hodnoty proměnné `#{flow}` do kulatých závorek, docílíme po dosazení do rovnice hladiny správného zápisu `-(-10)`, jenž dokáže metoda `evaluate` správně vyhodnotit.

Dalším problémem, který se během implementace vyskytl, byl formát desetinných čísel. V situaci, kdy byly nastaveny vysoké hodnoty parametru dt (kroky simulace byly rozděleny na velmi malé mezikroky) a výsledné hodnoty prvků byly tímto parametrem děleny, mohlo vzniknout číslo s velkým počtem desetinných míst, které metoda `evaluate` nebyla schopna zpracovat. K vyřešení této nesnáze, bylo použito formátování čísel se zaokrouhlením na zvolený počet desetinných míst. Tím bylo zamezeno vzniku čísel s příliš velkým počtem desetinných míst a výsledné hodnoty simulace jsou lépe čitelné. V našem případě bylo zvoleno zaokrouhlování na čtyři desetinná místa, avšak je možné zvolit i přesnější zaokrouhlování. Tímto krokem může dojít k menšímu rozdílu mezi správnými výsledky a těmi ze simulačního engine, který je ale zanedbatelný a velikost rozdílu bude záležet na volbě počtu desetinných míst použitých pro zaokrouhlování.

5.5.5 Testování a ladění simulačního engine

Nedílnou součástí každého vývoje software je taktéž průběžné testování a ladění. Při vývoji tohoto simulačního engine byly pro ověření správné funkčnosti použity unit testy a během integrace s GUI byla pro testování požadovaného chování vytvořena jednoduchá aplikace, která měla simulovat ovládací prvky na straně uživatelského rozhraní.

Pro unit testy slouží samostatná třída `SimulationTest`, která obsahuje několik testovacích případů, pomocí kterých byla ověřována správnost výsledných hodnot pokaždé, když došlo ve vyvíjeném simulačním engine ke změnám. Bližší popis jednotlivých testovacích případů je uveden dále v textu v kapitole 5.5.6.

Kromě verifikace správnosti výsledných hodnot simulovaných modelů bylo potřeba otestovat chování simulačního engine, jenž bylo definováno a požadováno ze strany uživatelského rozhraní. Pro tyto účely vznikl samostatný projekt simulující komunikaci mezi GUI a simulačním engine a také všechny požadované funkční prvky, kterými je možné simulaci ovládat. Tento projekt sloužil ryze pro pracovní účely během vývoje, a proto se jím nebudeme v této práci více zabývat. Pro zajímavost a lepší představu o tom, jak testování probíhalo, je součástí přílohy A.

5.5.6 Testovací případy unit testů

V rámci testování vyvíjeného simulačního engine bylo vytvořeno několik modelů systémové dynamiky, jež nám umožnily pomocí unit testů ověřit správnost výpočtů prováděných v simulačním engine. Jednotlivé testovací případy reprezentují základní konstrukce, které jsou v modelech systémové dynamiky velmi často využívány. Jelikož je obtížnější si z programového zápisu konkrétní model představit, uveďme stručný popis každého z nich.

exponentialGrowth

Tento test reprezentuje model exponenciálního růstu. S bližším popisem této konstrukce jsme se již seznámili v kapitole 3.3.1. Zachycuje populaci uživatelů libovolného software, která je rozšiřována na základě růstového faktoru, pod nímž bychom si mohli představit např. doporučení mezi uživateli či využití reklamy. Platí zde tedy, že čím větší je uživatelská základna prezentovaného software, tím rychleji k ní budou přibývat noví uživatelé.

exponencialDecay

ExponencialDecay je testovací případ, jež představuje model odpovídající chování exponenciálního klesání. Také s tímto typem zpětné vazby jsme se seznámili již dříve v kapitole 3.3.2. V tomto případě model simuluje opravování chyb vzniklých během vývoje software, přičemž oprava je ovlivňována výší hodnoty průměrné doby potřebné na opravu.

oscilation

Oscilace je vzor chování, jenž se snaží dosáhnout požadovaného cíle, ale díky zpoždění v systému ho nedosáhne, protože jakmile se k němu přiblíží, mine jej. Neustálá snaha o dosažení cílového stavu způsobuje, že systém kolem něj začne oscilovat. Náš testovací případ představuje

jednoduchý model uspokojení poptávky. Systém funguje tak, že pokud je k dispozici dostatek vyráběných produktů, firma propouští zaměstnance, jakmile jejich počet klesne pod určitou hranici, opět začne zaměstnance nabírat. Tento cyklus se opakuje stále dokola, jelikož nikdy nebude dosaženo vyváženého stavu, kdy by určitý počet zaměstnanců dokázal vyrobit takové množství produktů, jež by uspokojily poptávku, aniž by se snížilo požadované množství produktů na skladě.[41]

delay

Tento testovací případ reprezentuje model zpoždění s dosažením cílového stavu. V našem případě byl vytvořen model najímání nových zaměstnanců, kdy je stanoven požadovaný konečný stav. Jeho dosažení je ovlivňováno zpožděním, které udává, jak dlouho trvá proces přijetí jednoho zaměstnance. Jakmile je dosaženo požadovaného stavu, systém se ustálí.[42]

coflow

Coflow je jednoduchý model, který si můžeme představit jako zřetězení dvou procesů, kdy jeden využívá toho druhého. V našem případě jsme vytvořili model generování chyb při procesu vývoje software. Tato konstrukce je nám již velmi dobře známá, jelikož jsme se s ní setkali již při tvorbě referenčního modelu.[43]

testIf

Tento testovací případ přímo nesouvisí se systémovou dynamikou, ale zaměřuje se pouze na ověření správné funkčnosti podmiňovací funkce `If()`, jež byla v rámci simulačního engine implementována. Jedná se o všeobecně známou notaci využívanou v programování. Zápis této funkce vypadá následovně: `If(podmínka, then, else)`. Pokud je podmínka vyhodnocena kladně provede se výraz v části `then`, v opačném případě se provede `else`.

testSoftwareChangeModel

Posledním z uvedených testovacích případů je námi vytvořený referenční model procesu změny software. Jako všechny výše uvedené testovací případy, také on posloužil k ověření správné funkčnosti celého simulačního engine, ale primárně se k testování nepoužívá. Tento model byl detailně popsán v kapitole 4.2.

---

[41] MADACHY, Raymond J. *Software Process Dynamics*. 1st printing, New Jersey: John Wiley & Sons, Inc., 2008. 601 s. ISBN 978-0-471-27455-1. s. 177.
[42] MADACHY, Raymond J. *Software Process Dynamics*. 1st printing, New Jersey: John Wiley & Sons, Inc., 2008. 601 s. ISBN 978-0-471-27455-1. s. 169.
[43] MADACHY, Raymond J. *Software Process Dynamics*. 1st printing, New Jersey: John Wiley & Sons, Inc., 2008. 601 s. ISBN 978-0-471-27455-1. s. 164.

5.6 Simulace referenčního modelu pomocí simulačního engine

5.6.1 Zápis referenčního modelu

Vytvořený simulační engine pro simulaci modelů vytvořených metodou systémové dynamiky, který byl popsán výše, splňuje všechny potřebné požadavky a funkce umožňující spuštění simulace referenčního modelu. Simulace slouží primárně k ověření správné funkčnosti našeho simulačního engine, a proto nebude model sestaven v uživatelském rozhraní, ale pouze jako jeden z unit testů ve třídě `SimulationTest`. Zápis a definici modelu znázorňuje tento fragment kódu:

```
final Simulation simulation = new Simulation();
simulation.setStartSimulation(0);
simulation.setStepSimulation(1);
simulation.setEndSimulation(351);

final Set<Stock> stocks = new HashSet<Stock>();
stocks.add(new Stock("Requirements",
      "#{requirements generation rate}-#{tasks schedule rate}", 0));
...
final Set<Flow> flows = new HashSet<Flow>();
flows.add(new Flow("tasks schedule rate", "If(#{Requirements} > 0,
      #{Tasks scheduling productivity}*#{Project managers} , 0)"));
...
final Set<Auxiliary> auxuliaries = new HashSet<Auxiliary>();
auxuliaries.add(new Auxiliary("Job size", "500"));
...
try {
  final Result result = simulation.play();
}
catch (final EvaluationException e) {
  e.printStackTrace();
}
```

Jak je patrné z uvedeného kódu, simulaci spustíme pomocí metody `play()`. V tomto případě proběhne celá simulace najednou a její výsledky budou reprezentovány v instanci třídy `Result`. Výsledné hodnoty vybraných proměnných této simulace zobrazuje tabulka Tab. 11.

Time (Day)	Scheduled tasks	rework design errors rate	Errors corrected in test	total effort rate	Total effort
0	0	0	0	0	0
25	52	4,7981	28,3778	18,0687	415,3309
50	102	4,8	82,0151	18,0716	867,099
75	152	4,8	137,0127	18,0718	1318,8925

100	112	4,8	192,1129	12,0719	1680,6886
125	12	4,8	247,2403	12,0719	1982,4861
150	0	0,0027	296,4987	3,9727	2106,5483
175	0	0	339,9947	3,972	2205,8534
200	0	0	383,2069	3,972	2305,1534
225	0	0	418,2702	1,5071	2375,0966
250	0	0	421,6549	1,5003	2412,6528
275	0	0	421,7934	1,5	2450,155
300	0	0	421,799	1,5	2487,655
325	0	0	421,799	1,5	2525,155
350	0	0	421,799	0	2547,655

Tab. 11: Výsledné hodnoty Simulace x provedené v simulačním engine

5.6.2 Porovnání výsledků s nástrojem Vensim

Z výše uvedené tabulky Tab. 11 je patrná menší odlišnost od výsledných hodnot simulace stejného modelu pomocí nástroje Vensim, jež byly prezentovány v kapitole 4.3.6. V průběhu všech testů nebyly nikdy zaznamenány žádné rozdíly mezi výslednými hodnotami obou simulačních nástrojů, a proto se zdálo, že výsledky mohou být ovlivněny zaokrouhlováním čísel v simulačním engine, které bylo implementováno až v samém závěru celé práce. Po bližším prozkoumání byla ale tato domněnka zavrhnuta a příčinu bylo potřeba hledat jinde. Jedinou možností, jak odhalit příčinu způsobující rozdíl hodnot se jevilo být detailní porovnávání výsledků všech prvků simulovaného modelu. Inkriminované místo, kde anomálie vznikala, bylo nakonec úspěšně nalezeno a ihned bylo zřejmé, co ji způsobilo. V modelu jsou v tocích základního řetězce podmínky hlídající stav hladiny, ze které odčerpávají a pokud její hodnota klesne na nulu, pak z nich přestanou dále odčerpávat. Toky v modelu simulovaném nástrojem Vensim přestaly odčerpávat vždy o jeden krok později oproti tokům v simulačním engine. Příčinou tohoto rozdílného chování byla chyba známá pod označením strojové epsilon, jež se vyskytuje u datových typů reprezentujících čísla pomocí pohyblivé řádové čárky. Jak je patrné z tabulky Tab. 12, znázorňujícího chybovou část výsledných hodnot obou simulačních nástrojů, ve Vensim došlo při odečítání hodnoty toku od hladiny k chybě, která způsobila špatné vyhodnocení podmínky pro ukončení odčerpávání. Tím byla hodnota toku od hladiny odečtena ještě v následujícím kroku, i když už se tak stát nemělo. Správné vyhodnocení podmínky představují výsledky získané simulací modelu v námi vytvořeném simulačním engine.

	Vensim			**Simulační engine**		
Step	**design review rate**	**Reviewed design**	**code modification rate**	**design review rate**	**Reviewed design**	**code modification rate**
209	0	9,60033	2,4	0	9,6	2,4
210	0	7,20033	2,4	0	7,2	2,4
211	0	4,80033	2,4	0	4,8	2,4
212	0	2,40033	2,4	0	2,4	2,4

213	0	0,0003229971	2,4	0	0	0
214	0	-2,39967	0	0	0	0

Tab. 12: Rozdíl ve výsledcích Simulace x v nástroji Vensim a v simulačním engine

Z výše uvedeného můžeme usuzovat, že testováním bylo potvrzeno, že celá implementace simulačního engine proběhla úspěšně a je zcela funkční.

6 Závěr

Shrneme-li fakta a závěry, k nimž jsme v jednotlivých kapitolách dospěli, nelze než dospět k názoru, že cíle, jež si tato práce kladla v úvodu, a samotné zadání, které autor pro zpracování této práce obdržel, byly splněny. V úvodní kapitole, jíž můžeme zpětně označit za teoretickou, bylo pádnými argumenty zdůvodněno, proč simulace nepochybně patří do běžného života, a k problematice simulování v systémové dynamice lze s jistotou říci, že tato simulační metoda má v oblasti vytváření simulačních modelů a následného provádění simulace své nezastupitelné místo. Historický exkurz pomohl pochopit širší souvislosti vzniku této simulační metody i milníky jejího uplatňování včetně jejího vstupu do oblasti softwarového inženýrství.

Deskripcí základních principů a prvků systémové dynamiky, jejichž vysvětlení v praxi se autor vždy pokoušel vztáhnout na oblast vývoje softwarového procesu či do příbuzné problematiky, byl vytvořen systematický přehled minima znalostí potřebného k vytvoření simulačního modelu. V navazující kapitole byla popsána správná metodika tvorby v rámci systémové dynamiky, z níž čtenáři mohou nepochybně čerpat při pokusech o vytvoření vlastního modelu.

Poslední, ryze praktická, kapitola byla popisem autorova snažení vytvořit funkční simulační engine, který by byl využitelný v praxi. Ke snazšímu uchopení jistě napomohlo propojení s uživatelským rozhraním, tudíž výsledkem celé práce autora i tvůrce onoho simulačního rozhraní byl ucelený simulační nástroj, jehož výsledky byly srovnatelné s běžně využívaným simulačním nástrojem Vensim. Za velký úspěch lze v tomto kontextu jednoznačně pokládat odstínění strojového epsilon, čímž dokonce došlo k upřesnění výsledků simulace ve vytvořeném simulačním enginu oproti výsledkům v profesionálním nástroji Vensim. V případě zájmu je tedy možné, že by se tento engine proto mohl stát prototypem pro další vývoj, který by mohl vyústit ve vytvoření simulačního nástroje na profesionální úrovni.

Coby přínos této práce může být nepochybně považován také podnět k vytváření simulačních nástrojů nových. Autorem zpracovaný návod k tvorbě knihovny v jazyce Java, jež poskytuje engine umožňující provádění simulací modelů, nastínil jeden z možných postupů k implementaci systémové dynamiky. Snahou bylo vytvořit funkční, jednoduchou a snadno použitelnou knihovnu, která bude disponovat všemi důležitými vlastnostmi pro simulaci modelů systémové dynamiky. Tuto knihovnu je možné dále využít při realizaci vlastního simulačního nástroje nebo z ní jen čerpat samotný přístup k systémové dynamice.

Nezbývá tedy, než doufat, že tato práce dopomůže k rozšíření povědomí o možnostech simulace metodou systémové dynamiky a vzbudí ve čtenářích zájem o kreativní přístup k simulačním a modelovacím nástrojům, který mohou zúročit nejen pro oblast zmiňované systémové dynamiky, ale i v jiných metodách simulace.

Literatura

Literární zdroje

KOLÁŘ, V. *FEM principy a praxe metody konečných prvků*. 1. vyd. Praha: Computer Press, 1997. 401 s. ISBN 80-7226-021-9.

Koncept diplomové práce: Melcr, Jiří. Simulace Systémové dynamiky v prostředí Eclipse RCP, 2012.

MADACHY, R. *A Software Project Dynamics Model For Process Cost, Schedule And Risk Assessment*. California: 1994. 125 s. Disertační práce na Faculty Of The Graduate School University Of Southern California. Vedoucí disertační práce Dr. Barry Boehm, Dr. Behrokh Khoshnevis, Dr. Eberhardt Rechtin.

MADACHY, Raymond J. *Software Process Dynamics*. 1st printing, New Jersey: John Wiley & Sons, Inc., 2008. 601 s. ISBN 978-0-471-27455-1.

MILDEOVÁ, S., VOJTKO, V. a kol. *Systémová dynamika*. 2. vyd. Praha: Oeconomica, 2008. 150 s. ISBN 9708-80-245-1448-2.

POSPÍŠIL, Zdeněk. Dynamické systémy a systémová dynamika. In *Dynamika systémů a udržitelnost*. 1. vyd. Brno: Nadace Partnerství, 2007. od s. 25-68, 44 s. Soubor učebních textů.

ŠUSTA, M., NEUMAIEROVÁ, I. *Cvičení ze systémové dynamiky*. 1. vyd. Praha: Oeconomica, 2004. 94 s. ISBN 80-245-0780-3.

ZELENÝ, S., MANNOVÁ, B. *Historie výpočetní techniky*. 1. vyd. Praha: Scientia, 2006. 183 s. ISBN 80-86960-04-8.

Elektronické zdroje

Apache Maven Project. Dostupné z: <http://maven.apache.org/>.

Eclipse. Dostupné z: <http://www.eclipse.org/org/>.

Java. Dostupné z: <http://java.com/en/about/>.

JEval (http://jeval.sourceforge.net/), 2008. Version 0.9.4 Beta

Naval Postgraduate School, Monterey, California. Dostupné z: <http://faculty.nps.edu/vitae/cgi-bin/vita.cgi?p=display_vita&id=1023567788>.

Object Management Group Business Process Model and Notation. Dostupné z: <http://www.bpmn.org/>.

Search Software Quality. Dostupné z: <http://searchsoftwarequality.techtarget.com/>.

Stuart Madnick. Dostupné z: <http://web.mit.edu/smadnick/www/home.html>.

System Dynamics. Dostupné z: <http://www.systemdynamics.org/DL-IntroSysDyn/start.htm>.

Ventana Systems, Inc. (http://www.vensim.com/), 2010. Vensim® PLE for Windows Version 5.10b.

Wikipedie. Dostupné z: <http://cs.wikipedia.org>.

A Obsah DVD

/aplikace – zdrojové soubory a knihovny simulačního engine

- **/dist** – distribuované výsledné knihovny simulačního engine
- **/libs** – knihovny potřebné pro práci se simulačním engine
- **/sdlibrary** – zdrojové soubory výsledné knihovny
- **/sdlibrarygui** – aplikace pro testovací účely komunikace s GUI

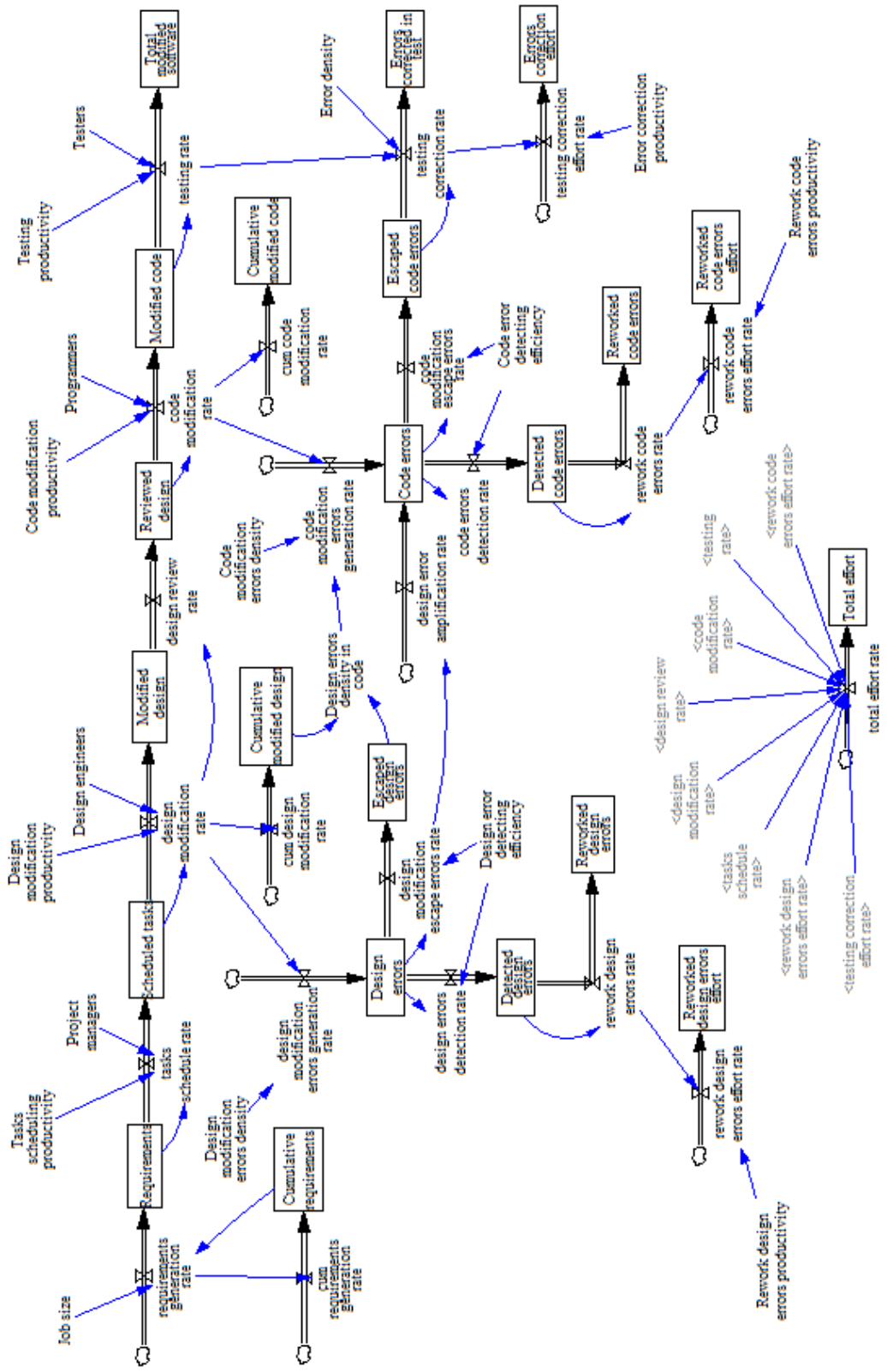
/referencni_model – referenční model vytvořený v nástroji Vensim PLE

/simulace – výsledky simulací referenčního modelu pomocí nástroje Vensim PLE a simulačního engine

/testovaci_pripady – modely testovacích případů v unit testech

/text_prace – soubory s textem této práce

B Obrázek referenčního modelu



C Rovnice referenčního modelu

```
(01)  Code error detecting efficiency=0.6
      Units: Detected Errors/Error

(02)  Code errors= INTEG (
            code modification errors generation rate+design error
amplification rate-
      code errors detection rate-code modification escape errors rate,
                  0)
      Units: Errors

(03)  code errors detection rate=Code errors*Code error detecting
efficiency
      Units: Errors/Day

(04)  Code modification errors density=1.5
      Units: Errors/Task

(05)  code modification errors generation rate=(Code modification errors
density+Design errors density in code)*code modification rate
      Units: Errors/Day

(06)  code modification escape errors rate=Code errors*(1-Code error
detecting efficiency)
      Units: Errors/Day

(07)  Code modification productivity=0.3
      Units: Tasks/Person

(08)  code modification rate= IF THEN ELSE(Reviewed design > 0, Code
modification productivity*Programmers
      , 0)
      Units: Tasks/Day

(09)  cum code modification rate=code modification rate
      Units: Tasks/Day

(10)  cum design modification rate=design modification rate
      Units: Tasks/Day

(11)  cum requirements generation rate=requirements generation rate
      Units: Requirements/Day

(12)  Cumulative modified code= INTEG (
            cum code modification rate,
                  0)
      Units: Tasks
```

```
(13)  Cumulative modified design= INTEG (
            cum design modification rate,
                  0)
      Units: Tasks

(14)  Cumulative requirements= INTEG (
            cum requirements generation rate,
                  0)
      Units: Requirements

(15)  Design engineers=5
      Units: Persons

(16)  design error amplification rate=    design modification escape
errors rate
      Units: Errors/Day

(17)  Design error detecting efficiency=0.8
      Units: Detected Errors/Error

(18)  Design errors= INTEG (
            design modification errors generation rate-design errors
detection rate-design modification escape errors rate
      ,
                  0)
      Units: Errors

(19)  Design errors density in code=Escaped design errors/IF THEN
ELSE(Cumulative modified design = 0, 1, Cumulative modified design
      )
      Units: Errors/Task

(20)  design errors detection rate=Design errors*Design error detecting
efficiency
      Units: Errors/Day

(21)  Design modification errors density=1.5
      Units: Errors/Task

(22)  design modification errors generation rate=Design modification
errors density*design modification rate
      Units: Errors/Day

(23)  design modification escape errors rate=  Design errors*(1-Design
error detecting efficiency)
      Units: Errors/Day

(24)  Design modification productivity=0.8
```

```
      Units: Tasks/Person

(25)  design modification rate=IF THEN ELSE(Scheduled tasks > 0, Design
engineers*Design modification productivity
      , 0)
      Units: Tasks/Day

(26)  design review rate=design modification rate
      Units: Tasks/Day

(27)  Detected code errors= INTEG (
            code errors detection rate-rework code errors rate,
                  0)
      Units: Errors

(28)  Detected design errors= INTEG (
            design errors detection rate-rework design errors rate,
                  0)
      Units: Errors

(29)  Error correction productivity=60
      Units: Effort/Error

(30)  Error density=0.08
      Units: Errors/Task

(31)  Errors corrected in test= INTEG (
            testing correction rate,
                  0)
      Units: Errors

(32)  Errors correction effort= INTEG (
            testing correction effort rate,
                  0)
      Units: Effort

(33)  Escaped code errors= INTEG (
            code modification escape errors rate-testing correction rate,
                  0)
      Units: Errors

(34)  Escaped design errors= INTEG (
            design modification escape errors rate,
                  0)
      Units: Errors

(35)  FINAL TIME  = 350
      Units: Day
      The final time for the simulation.
```

```
(36)  INITIAL TIME  = 0
      Units: Day
      The initial time for the simulation.

(37)  Job size=500
      Units: Requirements

(38)  Modified code= INTEG (
            code modification rate-testing rate,
                  0)
      Units: Tasks

(39)  Modified design= INTEG (
            design modification rate-design review rate,
                  0)
      Units: Tasks

(40)  Programmers=8
      Units: Persons

(41)  Project managers=2
      Units: Persons

(42)  Requirements= INTEG (
            requirements generation rate-tasks schedule rate,
                  0)
      Units: Requirements

(43)  requirements generation rate= IF THEN ELSE(Cumulative requirements
< Job size, Job size/10, 0)
      Units: Requirements/Day

(44)  Reviewed design= INTEG (
            design review rate-code modification rate,
                  0)
      Units: Tasks

(45)  rework code errors effort rate=     rework code errors rate/Rework
code errors productivity
      Units: Effort/Day

(46)  Rework code errors productivity=60
      Units: Effort/Error

(47)  rework code errors rate=Detected code errors*0.8
      Units: Errors/Day
```

```
(48)  rework design errors effort rate = rework design errors rate/Rework
design errors productivity
      Units: Effort/Day

(49)  Rework design errors productivity=60
      Units: Effort/Error

(50)  rework design errors rate=Detected design errors*0.3
      Units: Errors/Day

(51)  Reworked code errors= INTEG (
            rework code errors rate,
                  0)
      Units: Errors

(52)  Reworked code errors effort= INTEG (
            rework code errors effort rate,
                  0)
      Units: Effort

(53)  Reworked design errors= INTEG (
            rework design errors rate,
                  0)
      Units: Errors

(54)  Reworked design errors effort= INTEG (
            rework design errors effort rate,
                  0)
      Units: Effort

(55)  SAVEPER  =
              TIME STEP
      Units: Day [0,?]
      The frequency with which output is stored.

(56)  Scheduled tasks= INTEG (
            tasks schedule rate-design modification rate,
                  0)
      Units: Tasks

(57)  tasks schedule rate = IF THEN ELSE(Requirements > 0, Tasks
scheduling productivity*Project managers
       , 0)
      Units: Tasks/Day

(58)  Tasks scheduling productivity=      3
      Units: Tasks/Person

(59)  Testers=5
```

```
      Units: Persons

(60)  testing correction effort rate=testing correction rate/Error
correction productivity
      Units: Effort/Day

(61)  testing correction rate=Escaped code errors*testing rate*Error
density
      Units: Errors/Day

(62)  Testing productivity=
            0.3
      Units: Tasks/Person

(63)  testing rate=IF THEN ELSE(Modified code > 0, Testing
productivity*Testers, 0)
      Units: Tasks/Day

(64)  TIME STEP  = 1
      Units: Day [0,?]
      The time step for the simulation.

(65)  Total effort= INTEG (
            total effort rate,
                  0)
      Units: Effort

(66)  total effort rate = code modification rate+design modification
rate+tasks schedule rate+design review rate    +testing rate+rework code
errors effort rate+rework design errors effort rate+testing correction
effort rate
      Units: Effort/Day

(67)  Total modified software= INTEG (
            testing rate,
                  0)
      Units: Tasks
```